THE UNIVERSITY OF CALGARY

Higher-Order Charity

by

Marc A. Schroeder

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JULY, 1997

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Higher-Order Charity" submitted by Marc A. Schroeder in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor, Dr. Robin Cockett
Department of Computer Science

Dr. Lisa Higham
Department of Computer Science

Dr. Claude Laflamme
Department of Mathematics and Statistics

Date

Abstract

This thesis describes the higher-order Charity programming language which is an extension of first-order Charity. This results from extending the coinductive datatype definition mechanism to allow a new class of higher-order datatypes with parameterized destructors. This adds significant expressive power to the language. In particular it allows one to create "objects". The language is "higher-order" in the traditional sense that the exponential datatype can be defined, and so that functions can be treated as values.

The higher-order extension is traced from the extension of the syntax and the expressive gains delivered to the Charity programmer, down through the innards of the language and the modifications required in the implementation.

Acknowledgements

First, I would like to thank my colleagues, past and present, in the Charity Development Group: Tom Fukushima, Charles Tuckey, Peter Vesely, and Barry Yee. Without the stimulation and support from a core of talented researchers this work would not have been possible. A sincere thank-you to Ulrich Hensel as well, whose insightful input helped to shape much of this work.

I am extremely grateful to my friends and family for their help and understanding during my time at the University of Calgary. Especially to my mother Dianne Vallée, my brother Steve Schroeder, and Jennifer Wolfe.

Finally, many thanks to Dr. Robin Cockett for taking me on as his student, for his intellectual and financial support, and for introducing me to new ways of seeing my craft.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 What is Charity?

Charity is a *categorical* programming language which is *functional* in style. That is, the programming styles of both Charity and the various modern functional programming languages (such as Miranda[1] [23] or more recently Haskell [11]) are similar. However, Charity is based on a categorical semantics whereas functional programming languages are based on the lambda calculus. This means the fundamental operation in Charity is that of function *composition*, whereas in functional programming it is that of function *application*. This in turn means the exponential type (the type of functions) is primitive in functional programming languages, whereas it is not in Charity. Instead Charity takes as primitive the nullary and binary product types and provides a datatype definition mechanism.

The above distinction has an important consequence: in a functional programming language a function with input values of type A and output values of type B is itself a value, with type $A \to B$ —**the exponential type**[2]. Charity functions are not values. We say that

[1] Miranda is a trademark of Research Software Limited.

[2] It is often said that functions in functional programming languages are "first-class values": they have a type, they may be passed as input to and returned as output from other functions, etc., just like other values.

a language without the exponential datatype is **first-order**, and with it is **higher-order**.

To illustrate, consider the following function defined in Miranda:

```
double:: (* -> *) -> (* -> *)
double f = f . f
```

The `double` function takes a function `f` as input and returns a new function: the composite of `f` with itself[3]. Here `*` is a type variable. The type of the input is `* -> *`, and the type of the output is also `* -> *`, so `double` is a value of type `(* -> *) -> (* -> *)` overall. Note that `double` may be applied to itself:

```
quadruple = double double
```

The `double` function cannot be expressed in first-order Charity.

1.2 What is Higher-Order Charity?

The aim of this thesis is to describe the extension of first-order Charity to higher-order Charity. In the new language, as in the original, the exponential is *not* provided as primitive. Instead, the datatype definition mechanism is generalized such that a new class of **higher-order datatypes** may be defined. The exponential datatype is one of these, but more generally *objects* may be defined in the sense of *object-oriented programming*.

To give the reader a taste of higher-order Charity we show how the exponential datatype and the $double$ function can be defined:

$$\textbf{data}\ C \longrightarrow exp(A, B) = fn : C \longrightarrow A \Rightarrow B.$$

[3] In functional programming terminology, functions taking or returning other functions are called "higher-order functions", or "functionals".

$$\begin{array}{rcccl}\textbf{def}\ double: & exp(A,A) & \longrightarrow & exp(A,A) \\ = & (fn : f) & \mapsto & (fn : a \mapsto f\ f\ a).\end{array}$$

We also show how to define an object: a toy "turtle" like that of the logo language. First we need a direction datatype (dir) with an element for each direction (N, S, E, W):

$$\textbf{data}\ dir \longrightarrow C = N|S|E|W : 1 \longrightarrow C.$$

Next we need a $TURTLE$ datatype—the abstract class of turtles. A turtle can be told to *face* in a certain direction or to *advance* a number of steps:

$$\textbf{data}\ C \longrightarrow TURTLE = \left|\begin{array}{lcccccc} face & : & C & \longrightarrow & dir & \Rightarrow & C \\ adv & : & C & \longrightarrow & int & \Rightarrow & C. \end{array}\right.$$

Last we need a $turtle$ object, initially at the origin facing North. The turtle can be poked along by applying the $face$ and adv methods:

$$\begin{array}{rccl}\textbf{def}\ turtle: & 1 & \longrightarrow & TURTLE \\ = & () & \mapsto & \left(\left|(d,(x,y)) \mapsto \begin{array}{cccl} face & : & d & \mapsto (d,(x,y)) \\ adv & : & i & \mapsto \left\{\begin{array}{lcl} N & \mapsto & (d,(x,y+i)) \\ S & \mapsto & (d,(x,y-i)) \\ E & \mapsto & (d,(x+i,y)) \\ W & \mapsto & (d,(x-i,y)) \end{array}\right\} d \end{array}\right|\right)(N,(0,0)).\end{array}$$

1.3 The Structure of this Thesis

This thesis is divided into two parts: chapters 2—4 discuss the Charity language from the *user's* viewpoint, while chapters 5—7 examine the higher-order extension from the *implementor's* viewpoint, tracing its effects down through the various stages of the Charity interpreter. Specifically:

Chapter 2—An Overview of First-Order Charity We examine the Charity programming language before the higher-order extension.

Chapter 3—An Overview of Higher-Order Charity We discuss the generalization of the coinductive datatype definition mechanism and examine the resulting changes to the language. The exponential datatype and the process datatype, canonical examples, are defined.

Chapter 4—Using Higher-Order Charity The benefits of the higher-order extension are explored. This includes some uses of the exponential, the process datatype, and other higher-order datatypes. We discuss how to use higher-order Charity to define objects and simultaneously recursive functions.

Chapter 5—Variance The higher-order extension introduces "variance" into Charity. Variance is defined and variance analysis is discussed.

Chapter 6—Translation We examine the translation from the high-level Charity syntax as used by the programmer to the low-level representation as used by the Charity abstract machine.

Chapter 7—Compilation and Execution We introduce the Charity abstract machine and describe how it evaluates Charity programs.

Chapter 2

An Overview of First-Order Charity

The Charity programming language, before the higher-order extension, is referred to as **first-order Charity**. This chapter presents an overview of this language. Note that higher-order Charity is a seamless extension of the first-order fragment and so all the code presented here remains valid.

Charity is based on the theory of *strong categorical datatypes* [6, 7, 20], a modification of Hagino's *categorical datatypes* [9] which are related to the algebraic datatypes of modern functional programming languages. Unlike algebraic datatypes, however, the class of Charity datatypes is partitioned into two dual subclasses: the **inductive** datatypes and the **coinductive** datatypes. A formal, type-theoretic definition of Charity is presented in appendix A. This is the syntactic framework of the language, itself impotent without accompanying datatype definitions. Sections 2.1 and 2.2 describe first-order inductive and coinductive datatypes, respectively. Additional issues are covered in section 2.3. A catalogue of commonly used Charity datatypes is presented in appendix B. We will introduce the language after the higher-order extension in chapter 3.

2.1 Inductive Datatypes

The abstract syntax for inductive datatype definitions is

$$\textbf{data } L(A) \longrightarrow C = \left| \begin{array}{lllll} c_1 & : & F_1(A,C) & \longrightarrow & C \\ & \vdots & & & \\ c_n & : & F_n(A,C) & \longrightarrow & C. \end{array} \right.$$

In such a definition:

- L is the name of the datatype;
- A is a tuple $(A_1, \ldots, A_m)$ of type variables—the **parametric variables** ($m \geq 0$);
- C is a type variable—the **state variable**;
- each c_i is the name of a constructor ($1 \leq i \leq n$);
- each $F_i(A, C)$ is a type in terms of A and C.

Inductive definitions deliver the following:

1. a new m-ary *type constructor* L;
2. a set of new *constructors* (see below);
3. a set of new operations, or *combinators* (see below).

Inductive datatypes are also known as *left* datatypes, or *initial* datatypes.

One example is the datatype of finite lists, defined in terms of the nullary product datatype 1 and the binary product datatype $\_ \times \_$:

$$\textbf{data } list(A) \longrightarrow C = \left| \begin{array}{lllll} nil & : & 1 & \longrightarrow & C \\ cons & : & A \times C & \longrightarrow & C. \end{array} \right.$$

2.1.1 Constructors

As stated above, an inductive datatype definition delivers a set of constructors. To obtain the types of the constructors one simply replaces the state variable in the datatype definition with the datatype itself (ie. substitute $L(A)$ for C):

$$\begin{array}{ccccc} c_1 & : & F_1(A, L(A)) & \longrightarrow & L(A) \\ & \vdots & & & \\ c_n & : & F_n(A, L(A)) & \longrightarrow & L(A) \end{array}$$

Values of type $L(A)$ are built up by constructors: Given data of type A and, recursively, of type $L(A)$, we can apply constructor c_i to obtain data of type $L(A)$. Note that values of inductive datatypes must be finite.

The constructors for $list$ are

$$\begin{array}{ccccc} nil & : & 1 & \longrightarrow & list(A) \\ cons & : & A \times list(A) & \longrightarrow & list(A) \end{array}$$

so

$$cons(1, cons(2, cons(3, nil())))$$

is a value of type $list(int)$. Charity provides an alternative list syntax:

$$[1, 2, 3]$$

2.1.2 Inductive Combinators

In addition to the constructors, an inductive datatype definition delivers three operations over its values. These are the *fold*, the *case*, and the *map*. Note that fold is the fundamental operation, that the other two are special cases, and that they can be expressed as such. The case and the map are delivered as distinct operations due to their extreme usefulness and importance.

Combinator 1: Fold

Fold is the *destructive* operation for inductive datatypes: inductive values are constructed using constructors and are processed using the fold.

The type of fold is:

$$\mathit{fold}^L\{F_1(A,C) \longrightarrow C, \ldots, F_n(A,C) \longrightarrow C\} : L(A) \longrightarrow C$$

where each function between $\{$ and $\}$ is a *phrase* by which fold^L is parametrized. Note that the type of fold is given exactly by its corresponding datatype definition. A special "barbed wire" syntax is provided to the programmer:

$$\left\{\!\!\left|\begin{array}{ccccc} c_1 & : & v_1 & \mapsto & t_1 \\ & \vdots & & & \\ c_n & : & v_n & \mapsto & t_n \end{array}\right|\!\!\right\}$$

The behavior of fold is given by its defining *commuting diagrams*, one for each constructor:

$$\begin{tikzcd} F_i(A, L(A)) \arrow[r, "c_i"] \arrow[d, dotted, "F_i\{1_A, \mathit{fold}^L\{f_i\}\}"'] & L(A) \arrow[d, dotted, "\mathit{fold}^L\{f_i\}"] \\ F_i(A, C) \arrow[r, "f_i"'] & C \end{tikzcd}$$

The diagrams express the identities and uniqueness properties necessary to both reason about and evaluate fold expressions [25, 27]. Here, the identity states that constructing a value of type $L(A)$ and then destructing it is equal to recursively destructing its sub-components, and then applying the phrase. We draw a dotted map to express that it is unique. The uniqueness property states that given any h such that $c_i; h = F_i\{1_A, h\}; f_i$, then $h = \mathit{fold}^L\{f_i\}$.

The datatype of natural numbers is given by:

$$\textbf{data}\; nat \longrightarrow N = \left|\begin{array}{lcccl} zero & : & 1 & \longrightarrow & N \\ succ & : & N & \longrightarrow & N. \end{array}\right.$$

and has constructors:

$$\begin{array}{lcccl} zero & : & 1 & \longrightarrow & nat \\ succ & : & nat & \longrightarrow & nat \end{array}$$

(ie. $zero$, $succ\,zero$, $succ\,succ\,zero$, $\ldots$, are the values of type nat). We may use the fold to add natural numbers:

$$\textbf{def}\; add : nat \times nat \longrightarrow nat = (m, n) \mapsto \left\{\!\!\left|\begin{array}{lcccl} zero & : & () & \mapsto & n \\ succ & : & r & \mapsto & succ\, r \end{array}\right|\!\!\right\}\; m.$$

Note that fold^{nat} is essentially a `for` loop.

Fold is a generalization of the `fold` (or `foldr`) recursion functional for lists [4], and is often referred to as the *catamorphism* functional in the functional programming literature [13, 14]. We also say fold is the "structured recursion operator" for its corresponding datatype: one employs it to implement functions which recursively consume the values of that datatype.

Combinator 2: Case

Case is the nonrecursive specialization of fold. More intuitively, it is the conditional operation for inductive datatypes.

The type of case is:

$$case^L\{F_1(A, L(A)) \longrightarrow C, \ldots, F_n(A, L(A)) \longrightarrow C\} : L(A) \longrightarrow C$$

A special braced syntax is provided:

$$\left\{ \begin{array}{cccc} c_1 & : & v_1 & \mapsto & t_1 \\ & \vdots & & & \\ c_n & : & v_n & \mapsto & t_n \end{array} \right\}$$

The behavior of case is given by its defining commuting diagrams:

$$\begin{tikzcd} F_i(A, L(A)) \arrow[r, "c_i"] \arrow[dr, "f_i"'] & L(A) \arrow[d, dotted, "case^L\{f_i\}"] \\ & C \end{tikzcd}$$

As nat is recursive its main operation is the fold, but the case is also useful. Consider the $isZero$ function:

$$\textbf{def}\ isZero : nat \longrightarrow bool = n \mapsto \left\{ \begin{array}{ccc} zero & \mapsto & true \\ succ\ \_ & \mapsto & false \end{array} \right\} n.$$

The datatype of booleans is given by:

$$\textbf{data}\ bool \longrightarrow B = \left| \begin{array}{ccccc} true & : & 1 & \longrightarrow & B \\ false & : & 1 & \longrightarrow & B. \end{array} \right.$$

Alternatively, a shorthand syntax may be used:

$$\textbf{data}\ bool \longrightarrow B = true \mid false : 1 \longrightarrow B.$$

As $bool$ is not recursive its main operation is the case. We can use it to express all the boolean operations, such as not and or:

$$\textbf{def}\ not : bool \longrightarrow bool = b \mapsto \left\{ \begin{array}{ccc} true & \mapsto & false \\ false & \mapsto & true \end{array} \right\} b.$$

$$\begin{aligned}\textbf{def}\ or \quad &: \quad bool \times bool \longrightarrow bool \\ &= \quad (b_0, b_1) \mapsto \left\{ \begin{array}{lcl} true & \mapsto & true \\ false & \mapsto & \left\{ \begin{array}{lcl} true & \mapsto & true \\ false & \mapsto & false \end{array} \right\} b_1 \end{array} \right\} b_0.\end{aligned}$$

One may express these functions more concisely using pattern matching (section 2.3.1). Note that $case^{bool}$ is more commonly known as the `if-then-else` conditional.

Combinator 3: Map

Map is another specialization of fold, used to "lift" a function:

$$f : A \longrightarrow B$$

to a function over a parametric inductive datatype L. Specifically, the type of map is:

$$map^L\{A \longrightarrow B\} : L(A) \longrightarrow L(B)$$

A special syntax is provided:

$$L \left\{ \begin{array}{ccc} & \vdots & \\ v_h & \mapsto & t_h \\ & \vdots & \end{array} \right\}$$

The behavior of map is given by its defining commuting diagrams:

$$\begin{tikzcd} F_i(A, L(A)) \arrow[r, "c_i"] \arrow[d, dotted, "F_i\{f, map^L\{f\}\}"'] & L(A) \arrow[d, dotted, "map^L\{f\}"] \\ F_i(B, L(B)) \arrow[r, "c_i"'] & L(B) \end{tikzcd}$$

For example:

$$list\,\{isZero\}\;[succ\,zero, zero, succ\,succ\,zero] \quad\rightsquigarrow\quad [\mathit{false}, true, \mathit{false}]$$

Map is a generalization of the `map` functional for lists [4].

2.2 Coinductive Datatypes

The abstract syntax for coinductive datatype definitions is:

$$\textbf{data}\; C \longrightarrow R(A) = \left| \begin{array}{lcccl} d_1 & : & C & \longrightarrow & F_1(A, C) \\ & \vdots & & & \\ d_n & : & C & \longrightarrow & F_n(A, C). \end{array} \right.$$

Coinductive definitions deliver the following:

1. a new m-ary *type constructor* R;

2. a set of new *destructors* (see below);

3. a set of new *combinators* (see below).

Coinductive datatypes are also known as *right* datatypes, or *final* datatypes.

The datatype of infinite lists is given by:

$$\textbf{data } C \longrightarrow \mathit{inflist}(A) = \left| \begin{array}{lcccl} \mathit{head} & : & C & \longrightarrow & A \\ \mathit{tail} & : & C & \longrightarrow & C. \end{array} \right.$$

2.2.1 Destructors

To obtain the types of the destructors one simply replaces the state variable in the datatype definition with the datatype itself:

$$\begin{array}{lcccl} d_1 & : & R(A) & \longrightarrow & F_1(A, R(A)) \\ & \vdots & & & \\ d_n & : & R(A) & \longrightarrow & F_n(A, R(A)) \end{array}$$

Values of type $R(A)$ are broken down by destructors: given data of type $R(A)$, we can apply destructor d_i to obtain data of type A and, recursively, of type $R(A)$. Note that values of coinductive datatypes may be infinite.

The destructors for $\mathit{inflist}$ are

$$\begin{array}{lcccl} \mathit{head} & : & \mathit{inflist}(A) & \longrightarrow & A \\ \mathit{tail} & : & \mathit{inflist}(A) & \longrightarrow & \mathit{inflist}(A) \end{array}$$

2.2.2 Coinductive Combinators

In addition to the destructors, a coinductive datatype definition delivers three operations over its values. These are the *unfold*, the *record*, and the *map*. Here, the unfold is the fundamental operation and the other two are special cases, but they are delivered as distinct operations due to their usefulness.

Combinator 1: Unfold

Unfold is the *constructive* operation for coinductive datatypes: coinductive values are constructed using unfold and are processed by applying destructors to them. Unfold is the structured recursion operator for its corresponding datatype.

The type of unfold is:

$$\mathit{unfold}^R\{C \longrightarrow F_1(A,C),\ldots,C \longrightarrow F_n(A,C)\} : C \longrightarrow R(A)$$

Note that the type of unfold is given exactly by its corresponding datatype definition. A special "banana" syntax is provided to the programmer (note that the v, common to each phrase, has been factored out—the corresponding t_i are referred to as the *threads* of the unfold):

$$\left(\!\!\left|\; v \mapsto \begin{array}{ccc} d_1 & : & t_1 \\ & \vdots & \\ d_n & : & t_n \end{array} \;\right|\!\!\right)$$

The behavior of unfold is given by its defining commuting diagrams, one for each destructor:

$$\begin{tikzcd} R(A) \arrow[r, "d_i"] & F_i(A, R(A)) \\ C \arrow[u, dotted, "\mathit{unfold}^R\{f_i\}"] \arrow[r, "f_i"'] & F_i(A,C) \arrow[u, dotted, "F_i\{1_A, \mathit{unfold}^R\{f_i\}\}"'] \end{tikzcd}$$

As for fold, the diagrams express the identities and uniqueness properties necessary to both reason about and evaluate unfold expressions.

We can use the unfold to construct infinite lists, such as the infinite list of positive integers:

$$\textbf{def } ints : 1 \longrightarrow inflist(int) = () \mapsto \left(\!\left|\; i \mapsto \begin{array}{lcl} head & : & i \\ tail & : & i+1 \end{array} \right|\!\right) 0.$$

This evaluates to

$$\begin{array}{rcl} ints & \rightsquigarrow & (head : 0, tail : \ldots) \\ & \rightsquigarrow & (head : 0, tail : (head : 1, tail : \ldots)) \\ & \rightsquigarrow & (head : 0, tail : (head : 1, tail : (head : 2, tail : \ldots))) \\ & \vdots & \end{array}$$

where each subsequent step has to be explicitly requested by "poking" the unfold with destructors.

Unfold is a generalization of the `unfold` recursion functional for lists[1], and is often referred to as the *anamorphism* functional in the functional programming literature [13, 14].

Combinator 2: Record

Record is the nonrecursive specialization of unfold. More intuitively, it allows terms to be grouped into a structure. In practice, for nonrecursive coinductive datatypes it is the main constructive operation. For recursive datatypes it is used to construct a new value from an existing value.

The type of record is:

$$record^R\{1 \longrightarrow F_1(A, R(A)), \ldots, 1 \longrightarrow F_n(A, R(A))\} : 1 \longrightarrow R(A)$$

A special parenthesised syntax is provided:

[1] In Charity, *colists* as opposed to *lists* (see appendix B).

$$\left(\begin{array}{ccc} d_1 & : & t_1 \\ & \vdots & \\ d_n & : & t_n \end{array} \right)$$

The behavior of record is given by its defining commuting diagrams:

$$\begin{tikzcd}
R(A) \arrow[r, "d_i"] & F_i(A, R(A)) \\
1 \arrow[u, dotted, "record^R\{f_i\}"] \arrow[ur, "f_i"'] &
\end{tikzcd}$$

The datatype of triples is given by:

$$\textbf{data}\, T \longrightarrow triple(A, B, C) = \left| \begin{array}{llll} proj_0 & : & T & \longrightarrow & A \\ proj_1 & : & T & \longrightarrow & B \\ proj_2 & : & T & \longrightarrow & C. \end{array} \right.$$

As $triple$ is not recursive, its main operation is the record. We can use it to construct triples. We can then destruct them by applying projections, as in:

$$proj_1 \left(\begin{array}{lll} proj_0 & : & true \\ proj_1 & : & 42 \\ proj_2 & : & \text{"Deep Thought"} \end{array} \right) \rightsquigarrow 42$$

The above record syntax is equivalent to writing

$$(proj_0 : true,\ proj_1 : 42,\ proj_2 : \text{"Deep Thought"})$$

Combinator 3: Map

A map, of the same form as the map for inductive datatypes, is provided for coinductive datatypes:

$$map^R\{A \longrightarrow B\} : R(A) \longrightarrow R(B)$$

The behavior of this map is given by:

$$\begin{tikzcd}
R(B) \arrow[r, "d_i"] & F_i(B, R(B)) \\
R(A) \arrow[u, dotted, "map^R\{f\}"] \arrow[r, "d_i"'] & F_i(A, R(A)) \arrow[u, dotted, "F_i\{f, map^R\{f\}\}"']
\end{tikzcd}$$

2.3 Other Aspects of Charity

We conclude our overview of first-order Charity with a short survey of some other aspects: *pattern matching*, *combinators*, *context*, and *# and @*.

2.3.1 Pattern Matching

Much as do modern functional programming languages [17, 16, 23, 11, 18], Charity supports **pattern matching** [21, 22]. An abstraction

$$v \mapsto t$$

generalizes to a *patterned* abstraction

$$\left| \begin{array}{lcl} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_n & \mapsto & t_n \end{array} \right.$$

ie. a non-empty list of cases, where a case is a pattern-term pair. When applying a patterned abstraction, t_i is evaluated if and only if p_i is the first pattern which "matches" the input. Charity patterns must be *complete* in the sense that all functions must be total.

A canonical example of pattern matching is the boolean binary operator or. It was expressed without pattern matching in subsection 2.1. Note in the original version the programmer must explicitly decompose and test input using a tree of cases. Pattern matching makes case selection implicit. With pattern matching or simplifies to

$$\begin{array}{rcllcl} \textbf{def } or & : & bool \times bool & \longrightarrow & bool \\ & & & & \\ & = & (false, false) & \mapsto & false \\ & | & \_ & \mapsto & true. \end{array}$$

The example above illustrates pattern matching over pairs and inductive values. One can also pattern match over coinductive values using record patterns. For example, the following function tests whether all three elements of a triple are 0:

$$\begin{array}{rcllcl} \textbf{def } all\_0 & : & triple(int, int, int) & \longrightarrow & bool \\ & & & & \\ & = & (proj_0 : 0, proj_1 : 0, proj_2 : 0) & \mapsto & true \\ & | & \_ & \mapsto & false. \end{array}$$

2.3.2 Combinators

When defining combinators (functions), one may parametrize them by other combinator expressions. For example, consider the *filter* function over lists parametrized by an element testing predicate p:

$$\textbf{def}\ \mathit{filter}\{p : A \longrightarrow bool\} \quad : \quad list(A) \longrightarrow list(A)$$
$$= \quad l \mapsto \left\{ \begin{array}{lclcl} nil & : & () & \mapsto & [] \\ cons & : & (a,r) & \mapsto & \left\{ \begin{array}{lcl} true & \mapsto & cons(a,r) \\ false & \mapsto & r \end{array} \right\} p\ a \end{array} \right\} l.$$

Above, p is a *function variable*. When applied, *filter* is supplied with an actual *function*, as in:

$$\mathit{filter}\{isEven\}\ [1,2,3,4,5] \quad \rightsquigarrow \quad [2,4]$$

where

$$\textbf{def}\ isEven = x \mapsto eq_{int}(x \text{ mod } 2, 0).$$

2.3.3 Context

A term exists within a **context**: the scope of all variables bases (or patterns) which can bind its free variables. For this reason combinators accept a context σ as an additional input to be supplied to its parameters at application time. This detail is hidden from the programmer but it does affect the naive defining diagrams given in sections 2.1.2 and 2.2.2. The revised versions appear in figures 2.1—2.6 starting on page 20. The revised types appear below:

$$fold^L\{\ldots, F_i(A,C) \times \sigma \longrightarrow C, \ldots\} : L(A) \times \sigma \longrightarrow C$$
$$case^L\{\ldots, F_i(A,L(A)) \times \sigma \longrightarrow C, \ldots\} : L(A) \times \sigma \longrightarrow C$$
$$map^L\{A \times \sigma \longrightarrow B\} : L(A) \times \sigma \longrightarrow L(B)$$
$$unfold^R\{\ldots, C \times \sigma \longrightarrow F_i(A,C), \ldots\} : C \times \sigma \longrightarrow R(A)$$
$$record^R\{\ldots, \sigma \longrightarrow F_i(A,R(A)), \ldots\} : \sigma \longrightarrow R(A)$$
$$map^R\{A \times \sigma \longrightarrow B\} : R(A) \times \sigma \longrightarrow R(B)$$

Context passing, or *strengthening*, is a central aspect of the theory underlying Charity. It is discussed from a formal standpoint in [6, 7, 20], and from an intuitive one in chapter 6.

$$
\begin{tikzcd}
F_i(A, L(A)) \times \sigma \arrow[r, "c_i \times 1"] \arrow[d, dotted, "\langle map^{F_i}\{p_0, fold^L\{f_i\}\}, p_1\rangle"'] & L(A) \times \sigma \arrow[d, dotted, "fold^L\{f_i\}"] \\
F_i(A, C) \times \sigma \arrow[r, "f_i"'] & C
\end{tikzcd}
$$

Figure 2.1: Fold with context.

$$
\begin{tikzcd}
F_i(A, L(A)) \times \sigma \arrow[r, "c_i \times 1"] \arrow[dr, "f_i"'] & L(A) \times \sigma \arrow[d, dotted, "case^L\{f_i\}"] \\
 & C
\end{tikzcd}
$$

Figure 2.2: Case with context.

$$
\begin{tikzcd}
F_i(A, L(A)) \times \sigma \arrow[r, "c_i \times 1"] \arrow[d, dotted, "map^{F_i}\{f, map^L\{f\}\}"'] & L(A) \times \sigma \arrow[d, dotted, "map^L\{f\}"] \\
F_i(B, L(B)) \times \sigma \arrow[r, "c_i"'] & L(B)
\end{tikzcd}
$$

Figure 2.3: Inductive map with context.

$$
\begin{tikzcd}
R(A) \arrow[r, "d_i"] & F_i(A, R(A)) \\
C \times \sigma \arrow[u, dotted, "unfold^R\{f_i\}"] \arrow[r, "\langle f_i, p_1 \rangle"'] & F_i(A, C) \times \sigma \arrow[u, dotted, "map^{F_i}\{p_0, unfold^R\{f_i\}\}"']
\end{tikzcd}
$$

Figure 2.4: Unfold with context.

$$
\begin{tikzcd}
R(A) \arrow[r, "d_i"] & F_i(A, R(A)) \\
\sigma \arrow[u, dotted, "record^R\{f_i\}"] \arrow[ur, "f_i"'] &
\end{tikzcd}
$$

Figure 2.5: Record with context.

$$
\begin{tikzcd}
R(B) \arrow[r, "d_i"] & F_i(B, R(B)) \\
R(A) \times \sigma \arrow[u, dotted, "map^R\{f\}"] \arrow[r, "d_i \times 1"'] & F_i(A, R(A)) \times \sigma \arrow[u, dotted, "map^{F_i}\{f, map^R\{f\}\}"']
\end{tikzcd}
$$

Figure 2.6: Coinductive map with context.

2.3.4 # and @

Charity supports two special "identifiers": the **# variable** inside *folds* and the @ **function** inside *unfolds*.

The # ("hash") allows one to read the value being destructed inside a fold. In functional programming terminology, it allows the expression of *paramorphisms* [13]. Consider the $dropwhile$ function which traverses a list from left to right, dropping elements until they no longer satisfy a predicate:

$$\textbf{def}\ dropwhile\ \{p_A : A \longrightarrow bool\} : list(A) \longrightarrow list(A)$$

$$= l \mapsto \left\{ \begin{array}{lcll} nil & : & () & \mapsto\ [] \\ & & & \\ cons & : & (a, l') & \mapsto\ \left\{ \begin{array}{lcl} true & \mapsto & l' \\ false & \mapsto & \# \end{array} \right\} p_A\ a \end{array} \right\} l.$$

At any point during a fold the # represents the value being destructed (by the fold *before* it has been recursively applied). Generally for an inductive datatype L, # has type $F_i(A, L(A))$ for each of the i phrases of the fold. Thus, in our example above, the fold proceeds through the list from left to right recursively dropping each value until the predicate fails. At this point we reach a base case and return whatever is left of the list being processed.

Dropwhile is expressible without # in Charity, but requires a more complicated fold state involving the product datatype.

The # allows one to cast the fold operation as the case operation—its non-recursive special case—simply. For example, casing over lists:

$$\left\{ \begin{array}{lcccc} nil & : & \_ & \mapsto & \# \\ cons & : & \_ & \mapsto & \# \end{array} \right\}$$

The @ ("at") allows one to write the value being constructed inside an unfold. It's function is dual to that of the #. Consider the $pushdown$ function which inserts a value into an ordered infinite list, preserving the ordering:

$$\textbf{def} \quad \mathit{pushdown}\{<_A : A \times A \longrightarrow \mathit{bool}\} : A \times \mathit{inflist}(A) \longrightarrow \mathit{inflist}(A)$$

$$= (a,l) \mapsto \left(\!\!\left| (a',l') \mapsto \begin{array}{lcl} \mathit{head} & : & \left\{ \begin{array}{lcl} \mathit{true} & \mapsto & a \\ \mathit{false} & \mapsto & a' \end{array} \right\} a <_A a' \\ \\ \mathit{tail} & : & \left\{ \begin{array}{lcl} \mathit{true} & \mapsto & @\,(\mathit{head} : a', \mathit{tail} : l') \\ \mathit{false} & \mapsto & (\mathit{head}\; l', \mathit{tail}\; l') \end{array} \right\} a <_A a' \end{array} \right|\!\!\right) (\mathit{head}\; l, \mathit{tail}\; l).$$

Here, once the value has been inserted into the infinite list, the remainder is simply the infinite list $(head : a', tail : l')$ and so we produce it directly instead of producing a new state and recursively applying the unfold. Naively, the type of @ may be considered to be $R(A) \longrightarrow C$ (although there is a complication).

Pushdown is expressible without @ in Charity, but requires a more complicated unfold state involving the *sum* datatype (or *coproduct* datatype—see appendix B).

Note that @ also allows one to cast unfolds as records—again its non-recursive special case. For example, the following is the infinite list $0, 0, 1, 2, 3, 4, 5, \ldots$:

$$\left(\!\!\left| () \mapsto \begin{array}{lcl} head & : & 0 \\ tail & : & @\; ones \end{array} \right|\!\!\right) ()$$

Besides the expressive gains delivered by # and @, each reduces the complexity of code and therefore some computational overhead. There is another efficiency issue: in the case of the # an optimizing translation could eliminate unnecessary recursion when a premature # base case is reached. The typing of # and @ is discussed in [24].

Chapter 3

An Overview of Higher-Order Charity

The Charity programming language, after the higher-order extension, is referred to as **higher-order Charity**. This extension is a generalization of the coinductive datatypes. As such, the presentation here follows that of section 2.2 in the previous chapter: section 3.1 describes higher-order coinductive datatype definitions, and is succeeded by sections 3.2 and 3.3 which discuss the corresponding destructors and combinators, respectively. Sections 3.4 and 3.5 discuss pattern matching and context issues. The exponential and process datatypes are used as running examples throughout, and more examples will be presented in the next chapter.

3.1 Higher-Order Coinductive Datatypes

The abstract syntax for higher-order coinductive datatype definitions is

$$\textbf{data}\ C \longrightarrow R(A) = \left|\begin{array}{lcl} d_i & : & C \longrightarrow F_i(A,C) \\ & \vdots & \\ d_j & : & C \longrightarrow E_j(A) \Rightarrow F_j(A,C). \end{array}\right.$$

In such a definition $E_j(A)$ is a type in terms of A. The syntax allows the introduction of destructors both of the original d_i form and of the new d_j form. All the first-order coinductive datatypes are maintained simply by not using the d_j form.

The $\Rightarrow$ syntax is syntactic sugar. In fact, the system reads

$$d_j : C \longrightarrow E_j(A) \Rightarrow F_j(A, C)$$

as

$$d_j : E_j(A) \times C \longrightarrow F_j(A, C)$$

The above syntax is used as it is more consistent with both the original coinductive datatype syntax and the higher-order unfold syntax introduced in section 3.3.

The first and simplest of the higher-order datatypes is the **exponential** datatype:

$$\textbf{data } C \longrightarrow exp(A, B) = fn : C \longrightarrow A \Rightarrow B.$$

The type $exp(A, B)$ is the type of total functions from type A to type B, and is often written as B^A or as $A \Rightarrow B$ in the literature.

Another higher-order datatype is the **process** datatype:

$$\textbf{data } C \longrightarrow proc(A, B) = pr : C \longrightarrow A \Rightarrow C \times B.$$

The type $proc(A, B)$ is the type of total processes with input space A, output space B, and state space C.

The exponential and particularly the process datatype are discussed in more detail in chapter 4.

3.2 Destructors

The destructors for a higher-order coinductive datatype are

$$\begin{aligned} d_i &: R(A) \longrightarrow F_i(A, R(A)) \\ &\vdots \\ d_j &: E_j(A) \times R(A) \longrightarrow F_j(A, R(A)) \end{aligned}$$

The main idea of the higher-order extension is that the coinductive datatypes are generalized such that destruction is parametrized.

The destructor for exp is

$$fn : A \times exp(A, B) \longrightarrow B$$

The fn destructor applies a function f of type $exp(A, B)$ to an input a of type A, yielding an output b of type B:

$$fn(a, f) \rightsquigarrow b$$

The destructor for $proc$ is

$$pr : A \times proc(A, B) \longrightarrow proc(A, B) \times B$$

Similarly to the above, pr applies a process to an input, yielding an output. Additionally the internal state of the process changes and the process evolves.

3.3 Combinators

Combinator 1: Unfold

The type of unfold for higher-order coinductive datatypes (ignoring context) is

$$\mathit{unfold}^R\{C \longrightarrow F_i(A,C), \ldots, E_j(A) \times C \longrightarrow F_j(A,C)\} : C \longrightarrow R(A)$$

The syntax is:

$$\left(\left| v \mapsto \begin{array}{ccc} d_i & : & t_i \\ & \vdots & \\ d_j & : & v_j \mapsto t_j \end{array} \right|\right)$$

Note that the thread associated with d_i is a *term* as before, while the thread associated with d_j is a *function*. The input to this function is supplied at destruct-time.

The commuting diagrams for the d_i are as given in chapter 2. The diagrams for the d_j are as given here:

$$\begin{tikzcd}
E_j(A) \times R(A) \arrow[r, "d_j"] & F_j(A, R(A)) \\
E_j(A) \times C \arrow[r, "f_j"'] \arrow[u, dotted, "1_{E_j(A)} \times \mathit{unfold}^R\{f_j\}"] & F_j(A,C) \arrow[u, dotted, "F_j\{1_A, \mathit{unfold}^R\{f_j\}\}"']
\end{tikzcd}$$

As exp is nonrecursive its unfold operation is equivalent to its record operation. However, $proc$ is recursive and provides an example. First, we need the "success-or-failure" datatype—the datatype of exceptions:

$$\begin{array}{lcccccc} \textbf{data}\ SF(A) \longrightarrow C = & \mathit{ff} & : & 1 & \longrightarrow & C \\ & ss & : & A & \longrightarrow & C. \end{array}$$

Now consider the following $delay$ function:

$$\textbf{def}\ delay : \quad int \longrightarrow proc(A, SF(A))$$
$$= \quad x \quad \mapsto \quad \left(\!\left| l \mapsto pr : a \mapsto \left\{ \begin{array}{lll} nil & \mapsto & ([], \mathit{ff}) \\ cons(a, as) & \mapsto & (as, a) \end{array} \right\} l +\!\!+ [ss\ a] \right|\!\right) \ rep(x, \mathit{ff}).$$

In the above code $rep : int \times A \longrightarrow list(A)$ is the repeat function which produces an x-element list of *ff*'s and $+\!\!+ : list(A) \times list(A) \longrightarrow list(A)$ is the append function which concatenates two lists. A process may be considered a function with memory. The $delay$ function builds a delay process of length x. The initial state of the delay is "empty", ie. it is stocked with *ff*'s. At destruct-time input is supplied at one end and output arrives at the other. The output must have first flowed through the delay. Note that the nil case above is never taken, but is required for completeness.

Combinator 2: Record

The type of record is

$$record^R\{1 \longrightarrow F_i(A, R(A)), \ldots, E_j(A) \longrightarrow F_j(A, R(A))\} : 1 \longrightarrow R(A)$$

The syntax is

$$\left(\begin{array}{lll} d_i & : & t_i \\ & \vdots & \\ d_j & : & v_j \mapsto t_j \end{array} \right)$$

Again, the commuting diagrams for the d_i are as given in chapter 2. The diagrams for the d_j are as given here:

$$\begin{tikzcd}
E_j(A) \times R(A) \arrow[r, "d_j"] & F_j(A, R(A)) \\
E_j(A) \times 1 \arrow[u, dotted, "1_{E_j(A)} \times record^R\{f_j\}"] \arrow[ur, "po; f_j"'] &
\end{tikzcd}$$

Specializing from above, we obtain the type and diagram for $record^{exp}$:

$$record^{exp}\{A \longrightarrow B\} : 1 \longrightarrow exp(A, B)$$

$$\begin{tikzcd}
A \times exp(A, B) \arrow[r, "fn"] & B \\
A \times 1 \arrow[u, dotted, "1_A \times record^{exp}\{f\}"] \arrow[ur, "po; f"'] &
\end{tikzcd}$$

The $record^{exp}$ combinator introduces values of type $exp(A, B)$—the combinator encapsulates a function $f : A \longrightarrow B$ as a term of type $exp(A, B)$.

Consider

$$\begin{array}{lcccl}
\textbf{def}\; prd : & & 1 & \longrightarrow & exp(nat, nat) \\
 & = & () & \mapsto & \left(fn : \left| \begin{array}{lcl} zero & \mapsto & zero \\ succ\, n & \mapsto & n \end{array} \right. \right)
\end{array}$$

Then

$$prd \;\rightsquigarrow\; (fn : \langle\text{function}\rangle)$$

and

$$fn(succ\, succ\, zero, prd) \quad \leadsto \quad succ\, zero$$

Combinator 3: Map

The map combinator is the one most heavily affected by the higher-order extension. We devote chapter 5 to this aspect of the extension—*variance*—but introduce it here.

Recall that for *first-order* Charity map lifts a function

$$f : A \longrightarrow B$$

to a function

$$map^R\{f\} : R(A) \longrightarrow R(B)$$

For *higher-order* Charity, generally, the situation is more complex: it lifts a *pair* of functions

$$f^+ : A \longrightarrow B$$
$$f^- : B \longrightarrow A$$

to a function

$$map^R\{f^+ : A \longrightarrow B \;\&\; f^- : B \longrightarrow A\} : R(A) \longrightarrow R(B)$$

The syntax is

$$R\left\{\begin{array}{c} \vdots \\ v_h^+ \mapsto t_h^+ \quad \& \quad v_h^- \mapsto t_h^- \\ \vdots \end{array}\right\}$$

The diagrams for the d_j are

$$\begin{tikzcd}
E_j(B) \times R(B) \arrow[rr, "d_j"] & & F_j(B, R(B)) \\
E_j(B) \times R(A) \arrow[u, dotted, "1_{E_j(B)} \times map^R\{f\}"] \arrow[r, "E_j\{f\}^- \times 1_{R(A)}"'] & E_j(A) \times R(A) \arrow[r, "d_j"'] & F_j(A, R(A)) \arrow[u, dotted, "F_j\{f, map^R\{f\}\}^+"']
\end{tikzcd}$$

where

$$f = f^+ : A \longrightarrow B \;\&\; f^- : B \longrightarrow A$$

The idea is that as destructors consume additional *input* (in the E_j component) as well as produce *output* (in the F_j component), we must both *preprocess* the input with f^- and *postprocess* the output with f^+ when mapping. As will be explained, the type of map^{exp} is

$$map^{exp}\{C \longrightarrow A, B \longrightarrow D\} : exp(A, B) \longrightarrow exp(C, D)$$

3.4 Higher-Order Patterns

The higher-order extension affects pattern matching in one important way: we may use **higher-order record patterns**. For example, consider the composition function which takes two first-class functions (with suitable types) and returns their first-class composite. *Without* higher-order record patterns we write

$$\begin{aligned}
\textbf{def}\ comp \quad &: \quad exp(A, B) \times exp(B, C) \longrightarrow exp(A, C) \\
&= \quad (f, g) \mapsto (fn : a \mapsto fn(fn(a, f), g)).
\end{aligned}$$

With them we simplify to:

$$\begin{aligned}
\textbf{def}\ comp \quad &: \quad exp(A, B) \times exp(B, C) \longrightarrow exp(A, C) \\
&= \quad ((fn : f), (fn : g)) \mapsto (fn : a \mapsto g\, f\, a).
\end{aligned}$$

Here f and g are *function variables*.

Generally, higher-order record patterns are of the form

$$(d_j : f)$$

where f is an identifier. Higher-order record patterns were developed with Charles Tuckey and the implementation details are given in [22].

3.5 Combinators with Context

The commutative diagrams for unfold, record, and map given in this chapter must deal with context. Figures 3.1—3.3 complete the naive diagrams given thus far.

$$
\begin{tikzcd}
E_j(A) \times R(A) \arrow[r, "d_j"] & F_j(A, R(A)) \\
E_j(A) \times (C \times \sigma) \arrow[u, dotted, "1_{E_j(A)} \times \mathit{unfold}^R\{f_j\}"] \arrow[r, "\langle f_j, p_1; p_1 \rangle"'] & F_j(A, C) \times \sigma \arrow[u, dotted, "\mathit{map}^{F_j}\{1_A, \mathit{unfold}^R\{f_j\}\}"']
\end{tikzcd}
$$

Figure 3.1: Higher-order unfold with context.

$$
\begin{tikzcd}
E_j(A) \times R(A) \arrow[r, "d_j"] & F_j(A, R(A)) \\
E_j(A) \times \sigma \arrow[u, dotted, "1_{E_j(A)} \times \mathit{record}^R\{f_j\}"] \arrow[ur, "\mathfrak{f}_j"']
\end{tikzcd}
$$

Figure 3.2: Higher-order record with context.

$$
\begin{tikzcd}
E_j(B) \times R(B) \arrow[r, "d_j"] & F_j(B, R(B)) \\
E_j(B) \times (R(A) \times \sigma) \arrow[u, dotted, "1_{E_j(B)} \times \mathit{map}^R\{f\}"] \arrow[r, "\langle\langle\langle p_0, p_1; p_1 \rangle; \mathit{map}^{E_j}\{f\}^-, p_1; p_0 \rangle; d_j, p_1; p_1 \rangle"'] & F_j(A, R(A)) \times \sigma \arrow[u, dotted, "\mathit{map}^{F_j}\{f, \mathit{map}^R\{f\}\}^+"']
\end{tikzcd}
$$

Figure 3.3: Higher-order map with context.

Chapter 4

Using Higher-Order Charity

In the previous chapter we defined two important higher-order datatypes: the exponential datatype of functions and the datatype of processes. These are two higher-order datatypes among many. In this chapter we continue to illustrate the higher-order extension and the expressive gains delivered by presenting more examples using these two datatypes, and also by introducing others.

First, the *process datatype* is studied in greater detail in section 4.1. Then it is shown how to express *stacks* and *queues* in higher-order Charity in section 4.2. Generally, the higher-order extension allows one to express *objects* in the sense of object-oriented programming. Processes, stacks, and queues are some specific examples. The correspondence between higher-order Charity and object-oriented programming is discussed in section 4.3. The exponential datatype may be used to write *simultaneously recursive functions*, and even to realize an efficiency gain. This is demonstrated in section 4.4. It is shown how to implement a simple *parser* using higher-order datatypes in appendix D.

4.1 Processes

In [12] a technique for modeling processes as circuits is described. A **circuit** is, informally, an object with input space A, output space B, and state space C, represented diagrammatically as:

$$A \succ \boxed{\; C \;} \prec B$$

Each circuit is provided with a method ρ:

$$\rho : A \times C \longrightarrow C \times B$$

That is, circuits model processes as they consume input, produce output, and have an internal state which allows them to evolve over time as they are invoked.

We can model processes in higher-order Charity the same way using the process datatype:

$$\textbf{data } C \longrightarrow proc(A, B) = pr : C \longrightarrow A \Rightarrow C \times B.$$

The above is reminiscent of the exponential. In fact, $proc$ is a generalization in that exp is $proc$ where C is specialized to 1. In other words, as exp is nonrecursive one may generalize it to $proc$ by adding a state C, thus making it recursive. We say that processes are functions "extended in time".

The following examples use the semicolon syntax which expresses the composition of two functions:

$$v \mapsto t; f \quad = \quad v \mapsto f\ t$$

Process building operations can be implemented using the unfold. For example, we can compose processes (ie. wire them in series) much as we can for functions:

$$\textbf{def}\ ser\ : proc(A,B) \times proc(B,C) \longrightarrow proc(A,C)$$

$$= z \mapsto \left(\!\!\left|\ (pr_1, pr_2) \mapsto \begin{array}{rlcl} pr: & a & \mapsto & pr(a, pr_1) \\ ; & (pr'_1, b) & \mapsto & (b, pr_2) \\ ; & (pr'_2, c) & \mapsto & ((pr'_1, pr'_2), c) \end{array} \right|\!\!\right)\ z.$$

We can parallelize processes (ie. wire them in parallel):

$$\textbf{def}\ par\ : proc(A,B) \times proc(C,D) \longrightarrow proc(A \times C, B \times D)$$

$$= z \mapsto \left(\!\!\left|\ (pr_1, pr_2) \mapsto \begin{array}{rlcl} pr: & (a,c) & \mapsto & (pr(a, pr_1), pr(c, pr_2)) \\ ; & ((pr'_1, b), (pr'_2, d)) & \mapsto & ((pr'_1, pr'_2), (b, d)) \end{array} \right|\!\!\right)\ z.$$

We can also define other "wirings": the identity $wire$, the $split$ for branching, the wire-pair $twist$, a multi-wire exchange ex, and a feedback loop fb.

$$\begin{array}{rccl} \textbf{def}\ wire: & 1 & \longrightarrow & proc(A,A) \\ = & () & \mapsto & (\!|\, () \mapsto pr : a \mapsto ((), a) \,|\!)\ () \end{array}$$

$$\begin{array}{rccl} \textbf{def}\ split: & 1 & \longrightarrow & proc(A, A \times A) \\ = & () & \mapsto & (\!|\, () \mapsto pr : a \mapsto ((), (a,a)) \,|\!)\ () \end{array}$$

$$\begin{array}{rccl} \textbf{def}\ twist: & 1 & \longrightarrow & proc(A \times B, B \times A) \\ = & () & \mapsto & (\!|\, () \mapsto pr : (a,b) \mapsto ((), (b,a)) \,|\!)\ () \end{array}$$

$$\textbf{def}\ ex\ : 1 \longrightarrow proc((A \times B) \times (C \times D), (A \times C) \times (B \times D))$$

$$= () \mapsto (\!|\, () \mapsto pr : ((a,b),(c,d)) \mapsto ((), ((a,c),(b,d))) \,|\!)\ ()$$

$$\textbf{def}\ \mathit{fb}\ : \mathit{proc}(A \times C, B \times C) \times C \longrightarrow \mathit{proc}(A, B)$$

$$= z \mapsto \left(\left| (p,c) \mapsto \begin{array}{rlcl} pr: & a & \mapsto & pr((a,c),p) \\ ; & (p',(\_,c')) & \mapsto & pr((a,c'),p') \\ ; & (p'',(b,c'')) & \mapsto & pr((p'',c''),b) \end{array} \right| \right) z.$$

The second step in *fb* allows a circuit to stabilize. We use the infix operators ; ; for ser and || for par. Now if we define a nor-gate:

$$\begin{array}{lcll} \textbf{def}\ nor: & 1 & \longrightarrow & proc(int \times int, int) \\ = & () & \mapsto & \left(\left| () \mapsto pr : \left| \begin{array}{lcl} (0,0) & \mapsto & ((),1) \\ \_ & \mapsto & ((),0) \end{array} \right. \right| \right) (). \end{array}$$

We can define a basic RS-flipflop:

$$\textbf{def}\ \mathit{flipflop}\ : 1 \longrightarrow proc(int \times int, int \times int)$$

$$= () \mapsto \mathit{fb}(ex; ; ((nor; ; split)||(nor; ; split)); ; ex; ; (wire||twist), (1,0)).$$

The above examples demonstrate that we can build complex processes from simpler ones according to a type discipline. It also shows that we can use higher-order Charity to model hardware.

Note that $proc$ is the datatype of total, deterministic processes. We can also define the datatype of partial processes (ie. processes that can terminate):

$$\textbf{data}\ C \longrightarrow Pproc(A,B) = Ppr : C \longrightarrow A \Rightarrow SF(C \times B).$$

and the datatype of nondeterministic processes (ie. processes that can evolve in more than one way):

$$\textbf{data}\ C \longrightarrow NDproc(A,B) = NDpr : C \longrightarrow A \Rightarrow list(C \times B).$$

4.2 Stacks and Queues

Stacks (LIFOs) and **queues** (FIFOs)[1] can be specified using the same higher-order datatype:

$$\textbf{data } C \longrightarrow storage(A) = \left| \begin{array}{lcl} write & : & C \longrightarrow SF(A) \Rightarrow C \\ read & : & C \longrightarrow SF(A) \times C. \end{array} \right.$$

The destructors delivered are

$$\begin{array}{l} write : SF(A) \times storage \longrightarrow storage \\ read : storage \longrightarrow SF(A) \times storage \end{array}$$

The idea is that one may $write$ an element to the stack/queue object thus obtaining a new one (the ff case empties the object), and one may $read$ an element from the object again obtaining a new one (the ff case indicates the object is empty).

Stacks and queues are implemented using different unfolds:

$$\begin{array}{rcl} \textbf{def } stack & : & 1 \longrightarrow storage(A) \\ & = & () \mapsto \left(\left| l \mapsto \begin{array}{lcl} write & : & \left| \begin{array}{lcl} \mathit{ff} & \mapsto & [] \\ ss\ a & \mapsto & cons(a, l) \end{array} \right. \\ \\ read & : & \left\{ \begin{array}{lcl} [] & \mapsto & (\mathit{ff}, []) \\ cons(a, l') & \mapsto & (ss\ a, l') \end{array} \right\} l \end{array} \right| \right) []. \end{array}$$

$$\begin{array}{rcl} \textbf{def } queue & : & 1 \longrightarrow storage(A) \\ & = & () \mapsto \left(\left| l \mapsto \begin{array}{lcl} write & : & \left| \begin{array}{lcl} \mathit{ff} & \mapsto & [] \\ ss\ a & \mapsto & l +\!\!+ [a] \end{array} \right. \\ \\ read & : & \left\{ \begin{array}{lcl} [] & \mapsto & (\mathit{ff}, []) \\ cons(a, l') & \mapsto & (ss\ a, l') \end{array} \right\} l \end{array} \right| \right) []. \end{array}$$

[1] This presentation is inspired by that in [26].

Note that even though $write$ and $read$ are dual (ie. their types are symmetric) it is not guaranteed that they are inverses. Such propositions about the sensible behaviour of implementations must be proven.

4.3 Objects: Towards Object-Oriented Programming

Objects, in the sense of object oriented programming [1], can be expressed in higher-order Charity. We have already seen some examples: turtles, processes, stacks, and queues. This section makes some general observations about the relationship between higher-order Charity and object-oriented programming.

All the preceding datatype definitions are *specifications* of abstract datatypes. Their values, as generated by unfolds, are *implementations*. Abstract datatype specifications are presented algebraically, and consist of three components:

1. the name of the *type*;
2. the typed *operations* for this type;
3. the *equations* these operations must satisfy.

A higher-order datatype definition declares the first two only. The unfold then defines how the first will be represented internally, and how the second will manipulate that concrete representation. The third component—the equations—are not formulated in Charity. Instead, it is the programmer's job to state them at the meta-level and then to prove that the implementation satisfies them.

Generally, the operations of algebraic specifications are not restricted in their typing. Higher-order datatypes *are* restricted, however, in that the state variable must occur exactly once in the domain of each destructor. For instance, one could not specify an abstract datatype of sets in which union and intersection were provided as operations, as each requires a pair of sets as input. However, the "simple" abstract datatypes specifiable via the higher-order datatype definition mechanism do represent a significant increase in expressive power as they correspond to *objects* in the sense of object oriented programming. The following table illustrates the relationship:

ADT		OOP
datatype	$\equiv$	abstract class
unfold (not applied)	$\equiv$	class
unfold (applied)	$\equiv$	object
state	$\equiv$	state
destructor	$\equiv$	method

That is, to define an (abstract) class we define a higher-order datatype. Objects are values of that type. There are two essential facets of objects: they have an internal state, and they are interacted with exclusively via their methods. The state of an object is simply the state over which we unfold, and the methods for querying and manipulating the object/state are destructors. The unfold ensures that the internal state is hidden, thus guaranteeing proper data abstraction and modularity. The one central difference between traditional object oriented programming and higher-order Charity is that we currently lack inheritance and a class hierarchy.

4.4 Simultaneously Recursive Functions

Recall that fold is the structured recursion operator for inductive datatypes: It is used to recursively process a finite value. Often, however, one wishes to recursively process *multiple* values *simultaneously*. In functional programming languages this is accomplished using general recursion. For example, consider the `min` function which computes the minimum of two natural numbers expressed in Miranda:

```
min:: num -> num -> num

min m n = min' m n
          where
                min' 0       y       = m
                min' x       0       = n
                min' (x + 1) (y + 1) = min' x y
```

This function recurses simultaneously over each of its two inputs until one reaches its base case. The first to "bottom out" is the minimum, so the computation will terminate in time proportional to the smaller.

The `min` function can also be expressed in first-order Charity but it can not use such a straightforward algorithm. This is because fold is *singly recursive*. Instead, one subtracts the second input from the first. If the result is zero then the first is smaller, otherwise the second is smaller. Of course, this subtraction is implemented via a fold and so it must arbitrarily choose which of the two numbers to fold over. If the larger is chosen then the computation will terminate in time proportional to the larger.

One can express **simultaneously recursive functions** in higher-order Charity with the help of exp. This technique was discovered independently by Meijer [15].

4.4.1 The Minimum of Two Natural Numbers

We implement the original `min` algorithm in higher-order Charity:

$$\textbf{def}\; min \;: nat \times nat \longrightarrow nat$$

$$= (m,n) \mapsto fn(n, \left\{\begin{array}{|llll|} zero & : & () & \mapsto \; (fn : \_ \mapsto m) \\ \\ succ & : & (fn : f) & \mapsto \; \left(fn : \begin{array}{|lll} zero & \mapsto & n \\ succ\, n' & \mapsto & f\, n' \end{array}\right) \end{array}\right\} m).$$

That is, we fold over m to produce a nested function—of type $exp(nat, nat)$—and apply that function to n. The n drives the nested function, removing a layer of nesting each time it is decremented. Which bottoms out first indicates which number is the smaller. Note that, essentially, we have solved the problem of simultaneous recursion versus singly recursive folds by currying.

Note also: the translation function (chapter 6) is currently unoptimized, so the fold eagerly computes the entire nested function before applying it to n, even though the $succ$ phrase contains an early base case. An optimized translation proposed by Robin Cockett as yet unimplemented will eliminate this problem, allowing the computation to terminate with time always proportional to the smaller input.

4.4.2 The "Zip" of Two Lists

The `min` function scales up, from nat to $list$, as the `zip` function [4]. This takes a pair of lists and returns a list of pairs, where the elements of the first two lists have been paired component-wise. The length of the resulting list is equal to the length of the shorter input list as extra unpairable components are dropped. Again, this operation is expressed using simultaneous recursion in a functional language and so can be expressed in higher-order Charity using the exponential:

$$
\begin{aligned}
&\textbf{def}\ zip\ : list(A) \times list(B) \longrightarrow list(A \times B)\\
&\quad = (l_1, l_2) \mapsto fn\left(l_2, \left\{\begin{array}{|llll} nil & : & () & \mapsto \ (fn : \_ \mapsto [\,])\\ \\ cons & : & (a, (fn : f)) & \mapsto \ \left(fn : \begin{array}{|lll} nil & \mapsto & [\,]\\ cons(b, l) & \mapsto & cons((a, b), f\ l)\end{array}\right)\end{array}\right\} l_1\right).
\end{aligned}
$$

4.4.3 Equality

The ability of higher-order Charity to express simultaneously recursive functions is not only useful for nat and $list$, but generally for all recursive inductive datatypes. Consider that two elements of any inductive datatype can be tested for structural equality. This test is a simultaneous recursion over the two elements. For instance, an inductive $tree$ datatype can be defined in Charity:

$$\textbf{data}\ tree(A) \longrightarrow C = \left| \begin{array}{lcll} leaf & : & A & \longrightarrow & C \\ node & : & C \times C & \longrightarrow & C. \end{array} \right.$$

We can test trees for equality as follows:

$$\textbf{def}\quad eq_{tree}\{eq_A : A \times A \longrightarrow bool\} : tree(A) \times tree(A) \longrightarrow bool$$

$$= (t_1, t_2) \mapsto fn(t_1, \left\{ \left| \begin{array}{lcll} leaf & : & a & \mapsto & \left(fn : \left| \begin{array}{lll} leaf\ a' & \mapsto & eq_A(a, a') \\ \_ & \mapsto & false \end{array} \right. \right) \\ \\ node & : & ((fn : f_l), (fn : f_r)) & \mapsto & \left(fn : \left| \begin{array}{lll} node(l, r) & \mapsto & and(f_l\ l, f_r\ r) \\ \_ & \mapsto & false \end{array} \right. \right) \end{array} \right. \right\} t_2).$$

Chapter 5

Variance

First-order Charity's type variables can only occur *covariantly*. The higher-order extension generalizes datatypes so that their parametric type variables may occur both covariantly and *contravariantly*. In this chapter we define these terms and discuss variance analysis.

The concepts of *duality* and *variance* come from category theory ([26, 3, 8, 19], etc.): datatypes are modeled by functors. Duality is a form of symmetry which manifests itself as variance in parametric datatypes, in this case the symmetry between *input* (contravariance) and *output* (covariance). Variance with respect to datatypes was studied by Hagino and used in his categorical programming language [9], and generally by the functional programming community [13, 14].

5.1 Variance Basics

In this section we explore the concept of variance starting with some examples.

5.1.1 Distinct Input/Output Type Variables

Consider the exponential datatype:

$$\textbf{data}\; C \longrightarrow \mathit{exp}(A,B) = \mathit{fn} : C \longrightarrow A \Rightarrow B.$$

The parametric type variable A occurs in an "input" position and the parametric type variable B occurs in an "output" position. That is, an A is *consumed* at destruct-time, while a B is *produced*.

How does one map over exp? Clearly one uses the map combinator:

$$\mathit{map}^{\mathit{exp}}\{\ldots\} : \mathit{exp}(A,B) \longrightarrow \mathit{exp}(C,D)$$

but how are the parameters filled in? Viewing a value of type $\mathit{exp}(A,B)$ as an object, we draw:

$$A \succ\!\boxed{\text{“function”}}\!\prec B$$

To map this value to a value of type $\mathit{exp}(C,D)$ we "wrap" it between *preprocessing* and *postprocessing* functions:

$$\begin{array}{rcl} \mathit{pre} & : & C \longrightarrow A \\ \mathit{post} & : & B \longrightarrow D \end{array}$$

to obtain:

$$C \succ\!\boxed{\xrightarrow{\mathit{pre}} A \succ\!\boxed{\text{“function”}}\!\prec B \xrightarrow{\mathit{post}}}\!\prec D$$

Thus, to map over exp one uses the map combinator:

$$\mathit{map}^{\mathit{exp}}\{C \longrightarrow A, B \longrightarrow D\} : \mathit{exp}(A,B) \longrightarrow \mathit{exp}(C,D)$$

which encapsulates the input function between preprocessing and postprocessing functions, yielding the output function.

Expressing the map as a record we can write:

$$\textbf{def}\ map\_exp\{pre, post\} = f \mapsto (fn : c \mapsto post\ fn(pre\ c, f)).$$

5.1.2 Nondistinct Input/Output Type Variables

Next, consider the storage datatype:

$$\textbf{data}\ C \longrightarrow storage(A) = \left|\begin{array}{lcl} write & : & C \longrightarrow SF(A) \Rightarrow C \\ read & : & C \longrightarrow SF(A) \times C. \end{array}\right.$$

Here, the parametric type variable A occurs in both an input position ($write$) and an output position ($read$).

To map over $storage$ one uses the map combinator:

$$map^{storage}\{A \longrightarrow B \,\&\, B \longrightarrow A\} : storage(A) \longrightarrow storage(B)$$

where its single parameter has *both* a preprocessing function *and* a postprocessing function. This combinator encapsulates the input stack/queue behind a read/write front-end.

Expressing the map as an unfold we can write:

$$\textbf{def}\ map\_storage\{prewrite, postread\} = s \mapsto \left(\left| s \mapsto \begin{array}{lcl} write & : & i \mapsto write(SF\{prewrite\}\ i, s) \\ read & : & SF\{postread\}\ read\ s \end{array} \right|\right)\ s.$$

Note that $postread$ and $prewrite$ need not be inverses.

5.1.3 Variance Generalized

A (strictly) output type variable is called a **covariant** parameter, a (strictly) input type variable is called a **contravariant** parameter, and an input/output type variable is called a **divariant** parameter. The last possibility is that a type variable is introduced but not used in a datatype definition. In this case it is neither an input nor an output variable and is called an **invariant** parameter. We denote these four possible variances using the symbols $+, -, *, ?$ respectively.

When a datatype R is defined **variance analysis** must be performed, as:

1. the type signature of map^R must be calculated, and

2. R is invalid if its state variable C occurs contravariantly or divariantly. Otherwise one could define such undesirable types as

   $$\textbf{data } C \longrightarrow foo = bar : C \longrightarrow exp(C, C).$$

   or, equivalently,

   $$\textbf{data } C \longrightarrow foo = bar : C \longrightarrow C \Rightarrow C.$$

   To see the problem consider the typings

   $$unfold^{foo}\{C \times C \longrightarrow C\} : C \longrightarrow foo$$

   $$bar : foo \times foo \longrightarrow foo$$

   When applying *bar* one must pass in a value of type foo, but this violates the hiding of the internal state C of the unfold. Additionally, the foo datatype models the untyped λ-calculus.

Formally, each type variable A which occurs in the definition of a datatype R is assigned a **variance** $\nu_R(A) \in \{+, -, *, ?\}$. The variance-arity, or **varity** of a datatype R states the assignment of variance to each of its parametric type variables. We write:

$$\mathcal{V}(R) = [\nu_R(A_1), \ldots, \nu_R(A_m)]$$

Variance analysis is the calculation of $\mathcal{V}(R)$ and $\nu_R(C)$.

5.2 Examples

Charity provides two fundamental builtin type constructors for finite products. We state the varity of each:

- $\mathcal{V}(1) = []$
- $\mathcal{V}(\_ \times \_) = [+,+]$

As stated above, input types are contravariant while output types are covariant. Naively, this says type variables are contravariant if they occur to the left of $\Rightarrow$, and are covariant otherwise. Thus one would expect the following varities:

- $\mathcal{V}(bool) = []$
- $\mathcal{V}(SF) = [+]$
- $\mathcal{V}(nat) = []$
- $\mathcal{V}(list) = [+]$
- $\mathcal{V}(inflist) = [+]$
- $\mathcal{V}(exp) = [-,+]$
- $\mathcal{V}(proc) = [-,+]$
- $\mathcal{V}(storage) = [*]$

Consider the datatype:

$$\textbf{data}\ C \longrightarrow strange(X,Y,Z) = str : C \longrightarrow exp(exp(X,Y),Z).$$

What is $\mathcal{V}(strange)$? Anything to the left of a $\Rightarrow$ would be in a position of negative $(-)$ variance. However, in this case everything is in a position of positive $(+)$ variance. *But*: $\mathcal{V}(exp) = [-, +]$. This means Z occurs in a position of positive variance, but $exp(X, Y)$ has been substituted into a position of *negative* variance. This has the effect of flipping its varity so that X sits in a position of positive variance, while Y sits in a position of negative variance. That is, $\mathcal{V}(strange) = [+, -, +]$.

5.3 Formalizing Variance

The **variance algebra** is the triple $(V, \cdot, \vee)$ where

- $V = \{+, -, *, ?\}$ is the **set of variances**;
- $(\cdot) : V \times V \longrightarrow V$ is the **substitution operation** given by:

$\cdot$	$?$	$+$	$-$	$*$
$?$	$?$	$?$	$?$	$?$
$+$	$?$	$+$	$-$	$*$
$-$	$?$	$-$	$+$	$*$
$*$	$?$	$*$	$*$	$*$

- $(\vee) : V \times V \longrightarrow V$ is the **join operation** given by:

$\vee$	$?$	$+$	$-$	$*$
$?$	$?$	$+$	$-$	$*$
$+$	$+$	$+$	$*$	$*$
$-$	$-$	$*$	$-$	$*$
$*$	$*$	$*$	$*$	$*$

Note:

- $(V, \cdot)$ forms a commutative monoid with identity $+$.
- $(V, \vee)$ forms a commutative monoid with identity $?$.
- V forms a lattice:

$$
\begin{tikzcd}
 & * & \\
+ \arrow[ur] & & - \arrow[ul] \\
 & ? \arrow[ul] \arrow[ur] &
\end{tikzcd}
$$

The substitution operation says how to "flip" a variance v_1 when it sits in a position of variance v_2. The join operation says how to combine variances v_1 and v_2 when a type variable occurs once in a position of v_1 variance, then again in a position of v_2 variance.

We lift the join operator to varities:

$$[v_1, \ldots, v_m] \vee [v'_1, \ldots, v'_m] = [v_1 \vee v'_1, \ldots, v_m \vee v'_m]$$

We may read a varity $\vec{v}$ at index h:

$$\vec{v}[h]$$

We may also update a varity with v at index h:

$$update(\vec{v}, h, v)$$

returning the new varity.

Given the generalized form of a coinductive datatype definition:

$$\textbf{data}\ A_0 \longrightarrow R(A_1,\ldots,A_m) = \left|\begin{array}{lcl} & \vdots & \\ d_j & : & A_0 \longrightarrow E_j(A_1,\ldots,A_m) \Rightarrow F_j(A_0,\ldots,A_m) \\ & \vdots & \end{array}\right.$$

the variance analysis algorithm is as shown in figure 5.1.

$$\mathcal{A}(R) = \bigvee_{j=1}^{n} \mathcal{C}(-, E_j, \overbrace{[?,\ldots,?]}^{m+1}) \vee \mathcal{C}(+, F_j, \overbrace{[?,\ldots,?]}^{m+1})$$

where

$$\begin{array}{lcl} \mathcal{C}(v, A_h, \vec{v}) & = & update(\vec{v}, h, v) \\ \mathcal{C}(v, T(T_1,\ldots,T_{m'}), \vec{v}) & = & \bigvee_{h'=1}^{m'} \mathcal{C}(v \cdot v_{h'}, T_{h'}, \vec{v}) \\ & & \text{where } \mathcal{V}(T) = [v_1,\ldots,v_{m'}] \end{array}$$

Figure 5.1: The variance analysis algorithm.

That is, $\mathcal{A}$ steps through each destructor checking its E_j component (if present) and F_j component. The E_j sits in a position of negative variance while the F_j sits in a position of positive variance. Initially we know nothing about the variances of their variables and set them to $[?,\ldots,?]$. The $\mathcal{C}$ function returns this vector, updated with the variances for each of the type's variables. Note that the vectors obtained from each E_j and F_j must be joined.

The $\mathcal{C}$ function descends recursively through a type expression and determines the variance of each of its variables. In the recursive case for the constant type T we retrieve its varity, then descend through each of its subexpressions. At each step T sits in a position of v variance, so each of its subexpressions sits in a position of $v \cdot v_{h'}$ variance. A join of the vectors obtained from each subexpression must then be formed. In the base case for the variable type A_h we set its variance to be that of the position in which this occurrence sits (later join operations may merge this variance with the variance of other occurrences).

Now R is valid if:

$$\mathcal{A}(R)[0] = ? \text{ or } \mathcal{A}(R)[0] = +$$

and:

$$\mathcal{V}(R) = [\, \mathcal{A}(R)[1], \,\ldots,\, \mathcal{A}(R)[m] \,]$$

Inductive datatype definitions also require variance analysis, and algorithm $\mathcal{A}$ extends to them in the obvious way: there is no E_j component to check.

5.4 Variance and the Map Combinator

Given a valid datatype R, the type signature of its map combinator is obtained from $\mathcal{V}(R)$:

$$map^R\{S_1, \ldots, S_m\} : R(A_1, \ldots, A_m) \longrightarrow R(B_1, \ldots, B_m)$$

where each type signature S_h is:

$$\begin{array}{ll} \_ & \text{if } v_h = ? \\ A_h \longrightarrow B_h & \text{if } v_h = + \\ B_h \longrightarrow A_h & \text{if } v_h = - \\ A_h \longrightarrow B_h \,\&\, B_h \longrightarrow A_h & \text{if } v_h = * \end{array}$$

and where

$$\mathcal{V}(R) = [v_1, \ldots, v_m]$$

When mapping the variance information indicates which phrase (the covariant or the contravariant) to apply. “Atomic” maps of “compound” types are expanded to “compound” maps of "atomic" types:

$$\left[map^{A_h} \{\ldots, f_h \mathbin{\&} g_h, \ldots\}^{+} \right] =$$
$$\quad f_h$$

$$\left[map^{A_h} \{\ldots, f_h \mathbin{\&} g_h, \ldots\}^{-} \right] =$$
$$\quad g_h$$

$$\left[\!\!\left[map^{T\,\overbrace{(\ldots, T_{h'}, \ldots)}^{arity\ T}} \{\ldots, p_h, \ldots\}^{v} \right]\!\!\right] =$$
$$\quad map^{T} \underbrace{\{\ldots, \left[map^{T_{h'}} \{\ldots, p_h, \ldots\}^{v} \right] \mathbin{\&} \left[map^{T_{h'}} \{\ldots, p_h, \ldots\}^{\mathit{flip}\ v} \right], \ldots\}}_{arity\ T}$$

where

$$\begin{aligned} \mathit{flip}\ + &= - \\ \mathit{flip}\ - &= + \end{aligned}$$

and

$$p_h = f_h \mathbin{\&} g_h$$

Chapter 6

Translation

Charity has three distinct notations, each residing at a different level of abstraction. The highest, as introduced in chapters 2 and 3, is the **extended term logic** which is useful for programming. Next, also previously discussed, is the **core term logic** which is used as an intermediary representation. The lowest is the **combinatory logic** which is useful for evaluation. In this chapter we discuss these representations and the translations between them.

6.1 The Extended and Core Term Logics, and the Translation Between Them

The core term logic is a special case of the extended term logic: it's the latter minus patterns. For the first-order fragment it's the notation as introduced in chapter 2 discounting section 2.3.1. For the higher-order extension it's the notation as discussed in chapter 3 discounting section 3.4. The core term logic is described in [7, 5]. The extended term logic was first proposed in [21] and is fully described in [22].

The translation from extended to core term logic is known as "pattern matching". As previously illustrated, one may express a Charity function as a list of cases where a case is an input pattern followed by a term. The case to be taken is the first one whose pattern matches the input. The pattern matching algorithm translates such "patterned" functions as a tree of nested case functions, thus making the case selection process explicit.

The higher-order extension has little effect on pattern matching and the precise algorithm is beyond the scope of this thesis. [22] provides a detailed description.

6.2 The Combinatory Logic

The term logic is a "variable-ful" notation. One could evaluate it directly, but then one would have to deal with variable substitution [2]. Instead, we translate the term logic to a "variable-less" notation—the combinatory logic. [6, 5, 20, 27, 25] also deal with Charity combinators.

A **combinator theory** consists of a system of *types*, a set of *atomic combinators*, a system for building *compound combinator* expressions, and a set of *equations* between combinator expressions.

6.2.1 Types

A set of type constructors (with fixed arities, or *kinds*) generates a system of types, ie. the set of terms of the free algebra. For example:

$$\begin{aligned} 1 &: \Omega^0 \longrightarrow \Omega \\ \_ \times \_ &: \Omega^2 \longrightarrow \Omega \\ list &: \Omega \longrightarrow \Omega \end{aligned}$$

generates types such as:

$$list(A) \times (1 \times list(B))$$

6.2.2 Atomic Combinators and Compound Combinator Expressions

A combinator is a function from one type to another, parametric about other functions. Compound combinator expressions are built from atomic combinators of the form:

$$c\{T_1 \longrightarrow T_1', \ldots, T_n \longrightarrow T_n'\} : T_0 \longrightarrow T_0'$$

where

- c is the combinator name,
- each T_i and T_i' is a type.

We write this as a formation rule[1]:

$$\frac{f_1 : T_1 \longrightarrow T_1', \ldots, f_n : T_n \longrightarrow T_n'}{c\{f_1, \ldots, f_n\} : T_0 \longrightarrow T_0'}$$

6.2.3 Charity's Combinator Theory: The Combinatory Logic

Charity's type system is generated from the fundamental type constructors for products: 1 and $\_ \times \_$, the builtins (eg. int), and the user-defined type constructors (eg. $list$). Charity's atomic combinators for manipulating values of these types are described in the following paragraphs.

The identity and composition combinators are delivered as follows:

$$\frac{A \text{ type}}{id\{\} : A \longrightarrow A}\text{identity}$$

[1] In fact, such a rule is introduced parametrically, meaning that the types may be polymorphic. The theory must therefore account for type variable substitution and specialization. To test whether a combinator expression is valid one would use the unification algorithm [24].

$$\frac{f : A \longrightarrow B, g : B \longrightarrow C}{comp\{f, g\} : A \longrightarrow C}\text{composition}$$

When writing 0-ary combinators such as id the empty braces may be dropped. Also, the $comp$ combinator may be written using the infix notation $f; g$.

id is the identity combinator which outputs its input, while $comp$ is the composition combinator which pipelines two composable combinator expressions. The id and $comp$ combinators satisfy the standard identity and associative laws.

Fundamental combinators for finite products (! for 1 and $pair$, p_0, and p_1 for $\_ \times \_$) are delivered as follows:

$$\frac{A \text{ type}}{!\{\} : A \longrightarrow 1}\text{voiding (0-tuple)}$$

$$\frac{f : C \longrightarrow A, g : C \longrightarrow B}{pair\{f, g\} : C \longrightarrow A \times B}\text{pairing (2-tuple)}$$

$$\frac{A \text{ type}, B \text{ type}}{p_0\{\} : A \times B \longrightarrow A}\text{0th projection}$$

$$\frac{A \text{ type}, B \text{ type}}{p_1\{\} : A \times B \longrightarrow B}\text{1st projection}$$

The $pair$ combinator may be written using the angle bracket syntax $\langle f, g\rangle$.

The voiding combinator ! is the 0-tuple constructing combinator, ie. the unique map from A to 1 which forgets its input. The pairing combinator $\langle \_, \_\rangle$ is the 2-tuple constructing combinator parametric about its two component building phrases, while p_0 and p_1 are the 2-tuple destructing combinators, ie. the component projections. The finite product combinators satisfy the standard universal properties for products.

Charity's fundamental combinators are summarized in table 6.1. The combinators and equations delivered with user-defined datatypes are as described in chapters 2 and 3 ($fold$,

$unfold$, etc.).

$id : A \longrightarrow A$
$! : A \longrightarrow 1$
$\langle C \longrightarrow A, C \longrightarrow B \rangle : C \longrightarrow A \times B$
$p_0 : A \times B \longrightarrow A$
$p_1 : A \times B \longrightarrow B$

Table 6.1: Fundamental combinators.

Each Charity program is expressible as a combinator expression. Consider, for example, the function

$$x \mapsto \left(\left| () \mapsto \begin{array}{ccc} headd & : & x \\ taill & : & () \end{array} \right) \right) ()$$

which produces an infinite list of input x. The state of the unfold is set to the 0-tuple. This program is expressed as a combinator expression as

$$\langle !, id \rangle; unfold^{inflist}\{p_1, !\}$$

Section 6.3 gives the derivation.

6.2.4 Context

A combinator is a function, and so takes input. Combinators with arity 1 or greater (ie. combinators with parameters, excluding $comp$, $pair$, and records) take a pair as input: the first component is an input value proper, and the second is a **context**. In the example above, $unfold^{inflist}$ is applied to the 0-tuple, so the first component of its input is $!$. It needs access to its context (x), so that is propagated inside via id in the second component. Note also that each phrase is a function, so it too takes a pair as input: a local context and the global context propagated in from outside. For example, the hd phrase acts on a pair: the

0-tuple provided locally and the x provided globally. To access the x it projects away the local component. To summarize: as combinators provide a variable-free notation, context must be explicitly propagated inside all combinators whose phrases might look up variable values in scope.

6.3 Translation from Core Term Logic to Combinatory Logic

The translation from core term logic to combinatory logic for first-order Charity is given in several places, including [7]. In this section we re-present it pursuant to the higher-order extension. First, for clarity and to explain the way context is handled, we discuss the translation of Charity's basic framework (see figure 6.1 on page 61).

Each rule (excepting possibly the last) does the obvious, introducing a fundamental combinator discussed in the previous section. The first takes a variable to itself via the identity combinator id. The second handles 0-tuples by the "forgetting" combinator !. The third decomposes pairs, accessing their variable components by projecting, while the next composes pairs. The second last handles function application for functions f, where f is a 0-arity named function, constructor, or destructor. The last, and most interesting, handles function abstraction. This case also serves as a review of variable elimination and of the manner in which this translation makes global and local context management explicit. We are translating term $\{v' \mapsto t'\}\ t$ in context v, so first translate input term t with respect to v and pair it with id. This has the following effect when evaluating: a context is generated before this pair combinator is encountered. The first component uses it to evaluate t and the second preserves it. Both are then passed on to the abstraction $v' \mapsto t'$. However, this abstraction may not only access the *local* context v', but also the *global* context v, and so is actually translated as $[\![(v', v) \mapsto t']\!]$ in anticipation of the pair. The context v is now preserved and forwarded inside the abstraction, along with the input proper, via this pair.

Having discussed the preliminaries we can now give the translation of the unfold, the record, and the map (figure 6.2). The unfold (in context σ) translates much as the abstraction does, but the input and context are passed into the $unfold^R$ combinator delivered with datatype R. Each phrase has access to the unfold state v and the context σ. Higher-order phrases have access to an additional input v_j. An important note: this input is the leftmost component of the triple, the state follows, and the context is the rightmost component. This is for scoping purposes and is due to the manner in which pairs are decomposed (see figure 6.1).

The record translation is a special case of the unfold translation: it is the non-recursive specialization and as such requires no state. Thus it requires no initial state, hence no input (ie. it is a *term* and not a function), so the context is forwarded directly without pairing.

The map translation is similar to the unfold translation, respecting variance.

For completeness, the remaining translation phrases for the fold, case, inductive map (figure 6.3), and combinator (figure 6.4) are given. In fact, this last case generalizes those preceding it.

We conclude with some example translations. The first is the derivation promised in section 6.2:

$$\begin{aligned}
\left[\!\!\left[x \mapsto \left(\left| () \mapsto \begin{array}{ccc} head & : & x \\ tail & : & () \end{array} \right) () \right]\!\!\right] &= \langle [\![x \mapsto ()]\!], id\rangle; unfold^{inflist}\{[\![((), x) \mapsto x]\!], [\![((), x) \mapsto ()]\!]\} \\
&= \langle !, id\rangle; unfold^{inflist}\{p_1; [\![x \mapsto x]\!], !\} \\
&= \langle !, id\rangle; unfold^{inflist}\{p_1; id, !\}
\end{aligned}$$

A subsequent phase of the Charity system optimizes the id out of the first phrase of the unfold [27].

$$
\begin{aligned}
[\![x \mapsto x]\!] &= id \\
[\![v \mapsto ()]\!] &= \; ! \\
[\![(v_0, v_1) \mapsto x]\!] &= p_i ; [\![v_i \mapsto x]\!] \text{ where } i = 0 \text{ if } x \text{ occurs in } v_0\text{, and } i = 1 \text{ otherwise} \\
[\![v \mapsto (t_0, t_1)]\!] &= \langle [\![v \mapsto t_0]\!] , [\![v \mapsto t_1]\!] \rangle \\
[\![v \mapsto f\ t]\!] &= [\![v \mapsto t]\!] \,; f \text{ where } f \text{ is a function symbol} \\
[\![v \mapsto \{v' \mapsto t'\}\ t]\!] &= \langle [\![v \mapsto t]\!] , id \rangle ; [\![(v', v) \mapsto t']\!]
\end{aligned}
$$

Figure 6.1: Core term logic to combinatory logic translation, part 1: basics.

$$
\left[\!\!\left[\sigma \mapsto \left(\left| v \mapsto \begin{array}{ccc} d_i & : & t_i \\ & \vdots & \\ d_j & : & v_j \mapsto t_j \end{array} \right. \right) t \right]\!\!\right] =
$$

$$
\langle [\![\sigma \mapsto t]\!] , id \rangle ; unfold^R \{ [\![(v, \sigma) \mapsto t_i]\!] , \ldots , [\![(v_j, (v, \sigma)) \mapsto t_j]\!] \}
$$

$$
\left[\!\!\left[\sigma \mapsto \left(\begin{array}{ccc} d_i & : & t_i \\ & \vdots & \\ d_j & : & v_j \mapsto t_j \end{array} \right) \right]\!\!\right] =
$$

$$
record^R \{ [\![\sigma \mapsto t_i]\!] , \ldots , [\![(v_j, \sigma) \mapsto t_j]\!] \}
$$

$$
\left[\!\!\left[\sigma \mapsto R \left\{ \begin{array}{ccc} & \vdots & \\ v_h \mapsto t_h & \& & v'_h \mapsto t'_h \\ & \vdots & \end{array} \right\} t \right]\!\!\right] =
$$

$$
\langle [\![\sigma \mapsto t]\!] , id \rangle ; map^R \{ \ldots , [\![(v_h, \sigma) \mapsto t_h]\!] \,\&\, [\![(v'_h, \sigma) \mapsto t'_h]\!] , \ldots \}
$$

Figure 6.2: Core term logic to combinatory logic translation, part 2: coinductive datatypes.

$$\left[\!\!\left[\sigma \mapsto \left\{\begin{array}{cccc} c_1 & : & v_1 & \mapsto & t_1 \\ & \vdots & & & \\ c_n & : & v_n & \mapsto & t_n \end{array}\right\} t\right]\!\!\right] =$$

$$\langle [\![\sigma \mapsto t]\!], id\rangle; fold^L\{[\![(v_1,\sigma) \mapsto t_1]\!], \ldots, [\![(v_n,\sigma) \mapsto t_n]\!]\}$$

$$\left[\!\!\left[\sigma \mapsto \left\{\begin{array}{ccc} c_1\, v_1 & \mapsto & t_1 \\ & \vdots & \\ c_n\, v_n & \mapsto & t_n \end{array}\right\} t\right]\!\!\right] =$$

$$\langle [\![\sigma \mapsto t]\!], id\rangle; case^L\{[\![(v_1,\sigma) \mapsto t_1]\!], \ldots, [\![(v_n,\sigma) \mapsto t_n]\!]\}$$

$$\left[\!\!\left[\sigma \mapsto L\left\{\begin{array}{ccc} & \vdots & \\ v_h \mapsto t_h & \& & v'_h \mapsto t'_h \\ & \vdots & \end{array}\right\} t\right]\!\!\right] =$$

$$\langle [\![\sigma \mapsto t]\!], id\rangle; map^L\{\ldots, [\![(v_h,\sigma) \mapsto t_h]\!] \,\&\, [\![(v'_h,\sigma) \mapsto t'_h]\!], \ldots\}$$

Figure 6.3: Core term logic to combinatory logic translation, part 3: inductive datatypes.

$$\left[\!\!\left[\sigma \mapsto c\left\{\begin{array}{ccc} v_1 & \mapsto & t_1 \\ & \vdots & \\ v_n & \mapsto & t_n \end{array}\right\} t\right]\!\!\right] =$$

$$\langle [\![\sigma \mapsto t]\!], id\rangle; c\{[\![(v_1,\sigma) \mapsto t_1]\!], \ldots, [\![(v_n,\sigma) \mapsto t_n]\!]\}$$

Figure 6.4: Core term logic to combinatory logic translation, part 4: combinators.

Next we illustrate a translation for a higher-order datatype. Consider the function which takes input x and returns a function, where this function takes input y and returns the pair (x, y):

$$\begin{aligned}
[\![x \mapsto (fn : y \mapsto (x, y))]\!] &= record^{exp}\{[\![(y, x) \mapsto (x, y)]\!]\} \\
&= record^{exp}\{\langle [\![(y, x) \mapsto x]\!], [\![(y, x) \mapsto y]\!]\rangle\} \\
&= record^{exp}\{\langle p_1; [\![x \mapsto x]\!], p_0; [\![y \mapsto y]\!]\rangle\} \\
&= record^{exp}\{\langle p_1; id, p_0; id\rangle\}
\end{aligned}$$

The optimization yields:

$$record^{exp}\{\langle p_1, p_0\rangle\}$$

Chapter 7

Compilation and Execution

Charity programs are executed by the **Charity abstract machine**. Previous versions of the machine executed combinators directly [5]. However, it was found that by first compiling combinators to an even lower level representation, computations could be performed much more quickly and simply [10, 27].

A detailed description of the Charity abstract machine and the compilation function from combinators to machine instructions for first-order Charity was the subject of [27]. In this chapter, we concentrate on the changes the higher-order extension necessitated in both *compilation* and *execution*: the former is treated in section 7.2, and the latter in section 7.3. As neither can be understood in isolation and without some grounding in the general operation of the machine, we first present a primer in section 7.1.

7.1 Overview of the Charity Abstract Machine

The Charity abstract machine consists of

- A **heap pointer** H with associated *value heap* for storing data;
- A **program counter** C with associated *code stream* for storing code;

- A **dump stack pointer** D with associated *dump stack* for storing temporary results, subroutine return addresses, etc.

The **machine state** is a triple $(H, C, D)^1$. A Charity expression compiles down into a stream of machine instructions, into which C is an index. Execution begins with C pointing to the start of the stream and ends with a $HALT$ instruction. Whichever value the heap pointer addresses when halting is the result of the expression's evaluation[2]. For larger computations execution may need to be suspended occasionally in order to garbage collect the heap. Note that, as the code stream is randomly indexable, the machine may execute the $GOTO$ instruction as well the $JUMP$ and RET instructions for subroutines during the course of execution.

The coinductive datatype operations—unfold, record, and map—are all treated uniformly. Each is compiled into a short sequence of instructions for building "record" heap values (rec's). These are values of the coinductive datatype. In this sense, the cosmetic differences between these three operations are eliminated (fold, case, and the inductive map are all also compiled in a uniform way).

The operation of the Charity abstract machine on rec's is **lazy**, and supports **sharing**. A rec is an n-element array of **closures**, one for each destructor of the n-destructor datatype. A closure is itself a (potential) value, but at rec creation time each closure is unevaluated. When destructor i is applied to rec it forces evaluation of closure i. This is called "poking". This closure is then updated with the resulting value so that subsequent pokes do not force redundant reevaluation, but rather return the precomputed result. In other words, a value of a coinductive datatype is a record whose fields are initially unaccessed and unevaluated. As we poke this structure by applying destructors, we access the corresponding fields, evaluate, and update, thus developing the value as needed (the *laziness*). If we reaccess a previously

[1]In fact, for technical reasons relating to garbage collection and space efficiency, the heap and dump are split into a number of specialized heaps and dumps, each for storing a specific kind of value. This does not affect our description.

[2]This value must be decompiled.

developed field, we don't force any reevaluation (the *sharing*). Sections 7.2 and 7.3 explain the above concepts further.

7.2 Compilation

In this section we give the compilation function from combinators to machine instructions, restricted to those cases affected by the higher-order extension. This section and the next also correct some errors in the original description for first-order Charity [27].

We begin with the compilation of destructor and record combinators as records are simpler than unfolds and maps, yet illustrate most of the issues in compiling all three. The relevant cases are given in figure 7.1.

$$
\begin{array}{llcl}
(C_{13a}) & [d_i] & = & DESTR\{i\} \\
(C_{13b}) & [d_j] & = & HODESTR\{j\} \\
(C_{14}) & [\![record^R\{f_1,\ldots,f_n\}]\!] & = & JUMP\{x\} \\
 & & & where \\
 & & & x := ALLOC\{n\}.BLDCLO\{1,x_1\}.\ldots.BLDCLO\{n,x_n\}.RET \\
 & & & \quad \begin{array}{lll} where \;\; x_i & := & BLDUPDATE \\ & . & [f_i] \\ & . & UPDATE\{i\} \\ & . & RET \\ x_j & := & [f_j] \\ & . & RET \end{array}
\end{array}
$$

Figure 7.1: Destructor and record compilation.

The original rule for destructors, C_{13}, has been generalized to C_{13a} and C_{13b}, where the former handles first-order destructors and the latter higher-order destructors. In fact, C_{13a} is the old rule and we are simply adding a new instruction, $HODESTR$, with the new rule. $HODESTR$ is the only addition to the instruction set which the higher-order extension necessitates! Each of the operations $DESTR$ and $HODESTR$ have, as operand, the destructor's position.

As stated in rule C_{14}, a record compiles as a jump to a subroutine beginning at label x ($JUMP$). This subroutine, when invoked, will allocate a new record value on the heap ($ALLOC$), initialize each of its n closures, (the n BLDCLO instructions), and return (RET). Each closure points to the start of a corresponding subroutine to be executed when it is poked. The subroutine is, essentially, a compiled phrase of the original record, but in the first-order case it is wrapped within the updating instructions ($BLDUPDATE$ and $UPDATE$). The first prepares for the update and the second carries it out, as explained in the next section. We must not update in the higher-order case as extra input will be supplied to the closure at poke time *each time we poke it*. That is, the closure is not an *expression*, but a *function* parametrized by an input. Thus it can never be reduced to one value, but must be reevaluated each time.

As explained in chapter 2, the operators delivered with Charity datatypes, most notably fold and unfold, are structured recursion operators. That is, the recursive nature of the operations remains *implicit* down to the combinator level. The compilation from combinators to machine instructions makes recursion *explicit*. Figure 7.2 shows how for unfold.

$$
\begin{array}{lll}
(C_{15}) \quad \llbracket unfold^R\{f_1,\ldots,f_n\}\rrbracket & = & JUMP\{x\} \\
& & where \\
& & x := ALLOC\{n\}.BLDCLO\{1,x_1\}.\ldots.BLDCLO\{n,x_n\}.RET \\
& & \quad \begin{array}{llll}
where & x_i & := & BLDUPDATE \\
& & . & \llbracket\langle f_i,p_1\rangle; map^{F_i}\{p_0, jump\{x\}\}\rrbracket \\
& & . & UPDATE\{i\} \\
& & . & RET \\
& x_j & := & \llbracket\langle f_j,p_1;p_1\rangle; map^{F_j}\{p_0, jump\{x\}\}\rrbracket \\
& & . & RET
\end{array}
\end{array}
$$

Figure 7.2: Unfold compilation.

Rule C_{15} is very similar to rule C_{14}, but differs in the compilation of the phrases f_i and f_j. In the first-order case f_i is executed in context, the result is paired with the context, and the unfold is mapped onto this result, making the recursion explicit. The higher-order case is nearly identical, but the additional input is taken into account when propagating the context to the map. Note that this rule simply restates the recursive definition of unfold as expressed in the commuting diagram of chapter 3 (as the rule for the record restates *its* commuting diagram). The main point of interest is that the recursive mapping of the unfold is via the label x.

Like rule C_{15} for unfold, rule C_{16} for map is derived from its diagrammatic definition. See figure 7.3. Here f is, in general, a set of map phrases $f_1^+ \& f_1^-, \ldots, f_m^+ \& f_m^-$—a positive and negative phrase for each parameter of a coinductive datatype R with arity m. Thus, map^{F_i}, map^{E_j}, and map^{F_j} must respect variance as described in chapter 5.

$$
\begin{array}{lll}
(C_{16}) \quad [\![map^R\{f\}]\!] & = & JUMP\{x\} \\
& & where \\
& & x := ALLOC\{n\}.BLDCLO\{1, x_1\}.\ldots.BLDCLO\{n, x_n\}.RET \\
& & \quad where \quad x_i \quad := \quad BLDUPDATE \\
& & \qquad\qquad\qquad\quad . \quad [\![\langle p_0; d_i, p_1 \rangle; map^{F_i}\{f, jump\{x\}\}^+]\!] \\
& & \qquad\qquad\qquad\quad . \quad UPDATE\{i\} \\
& & \qquad\qquad\qquad\quad . \quad RET \\
& & \qquad\qquad\quad x_j \quad := \quad [\![\langle\langle\langle p_0, p_1; p_1 \rangle; map^{E_j}\{f\}^-, p_1; p_0 \rangle; d_j, p_1; p_1 \rangle]\!] \\
& & \qquad\qquad\qquad\quad . \quad [\![map^{F_j}\{f, jump\{x\}\}^+]\!] \\
& & \qquad\qquad\qquad\quad . \quad RET
\end{array}
$$

Figure 7.3: Coinductive map compilation.

7.3 Execution

A **machine state transition** is a one-step updating of the machine state, one for each machine instruction. A computation is a sequence of transitions driven by a sequence of instructions. The machine transitions for First-Order Charity are given in [27]. The higher-order extension requires the modification of only one of those rules: rule 17. It is generalized to a case for first-order destructors (17a—the same as the original rule 17), and a case for higher-order destructors (17b). First, we present some other basic rules.

Rules 10 and 11 deal with subroutine calling and returning (table 7.1 on page 69). $JUMP$ lends control to a specified point and pushes the return address onto the dump stack. RET returns control and pops the return address off of the dump stack.

	H	C	D		H	C	D
10	v	$JUMP\{c'\}.c$	d	$\mapsto$	v	c'	$cont\{c\}.d$
11	v	$RET.c$	$cont\{c'\}.d$	$\mapsto$	v	c'	d

Table 7.1: Subroutine calling and returning.

	H	C	D		H	C	D
15	v	$ALLOC\{n\}.c$	d	$\mapsto$	$rec\{(\epsilon,\epsilon)\}_{i=1}^{n}.v$	c	d
16	$rec\{(\epsilon,\epsilon)\}_i.v$	$BLDCLO\{i,c_i\}.c$	d	$\mapsto$	$rec\{(v,c_i)\}_i.v$	c	d

Table 7.2: Record value construction.

Rules 15 and 16 construct record values on the heap (table 7.2). $ALLOC$ allocates an n-closure record, which the heap pointer H then addresses. The old value addressed by H is remembered for subsequent $BLDCLO$ instructions using an auxiliary machine register. Each $BLDCLO$ then initializes a given closure of the record. A closure is a pair: some *code* and a *value* to act on. The start address of the code is supplied by the $BLDCLO$, and the address of the value is supplied by the auxiliary register.

	H	C	D		H	C	D
17a	$v : rec\{(v_i,c_i)\}_i$	$DESTR\{i\}.c$	d	$\mapsto$	$v_i.v$	c_i	$cont\{c\}.d$
17b	$v : \langle e,w\rangle$ $w : rec\{(v_j,c_j)\}_j$	$HODESTR\{j\}.c$	d	$\mapsto$	$v' : \langle e,v_j\rangle.w$	c_j	$cont\{c\}.d$

Table 7.3: Record value destruction.

Rules 17a and 17b deal with destructor application (table 7.3). In the first-order case we are poking closure i of the record pointed to by H. Thus we execute closure code c_i at value v_i as a subroutine, pushing the return address onto the dump stack. Again, an auxiliary register is used to remember the record's address for the $BLDUPDATE$ instruction. The higher-order case is similar, but H addresses a pair, not a record. The first component of the pair is the input supplied at destruct time, e, and the second is the record to be destructed. This time we execute closure code c_j at value $\langle e, v_j \rangle$.

	H	C	D		H	C	D
18a	$v.w$	$BLDUPDATE.c$	d	$\mapsto$	v	c	$update\{w\}.d$
18b	v	$UPDATE\{i\}.c$	$update\{w\}.d$	$\mapsto$	v	c	d
	$w : rec\{(v_i, c_i)\}_i$				$w : rec\{v, RET\}_i$		

Table 7.4: Closure updating.

Last, rules 18a and 18b handle closure updating (table 7.4). $BLDUPDATE$ is executed at the start of closure execution, and pushes the address of the record being poked onto the dump stack. $UPDATE$ is executed at the end of closure execution, and updates that closure of that record in the following way: the result value is stored in the closure with an "empty" subroutine for evaluating it (ie. a single RET). Subsequent pokes of the same closure will simply return the precomputed value.

Chapter 8

Conclusion

In this final chapter we summarize our results, briefly discuss how they might be applied, and look towards the future.

8.1 Summary

In this thesis we have described the higher-order extension of first-order Charity: a generalization of its coinductive datatype definition mechanism by parameterizing destructors. Higher-order Charity realizes some expressive gains over its first-order predecessor, including:

- the ability to define the exponential datatype, rendering functions first-class values;
- the ability to define objects generally, opening the door to object-oriented programming in Charity;
- the ability to define simultaneously recursive functions, leading to a much more natural implementation of several common functions.

The effect of the extension on Charity syntax is slight, and is backwards compatible with first-order Charity. The effects on the various stages of Charity interpretation are also clean, as witnessed by the minimal updates required in the compilation and execution phases.

8.2 Future Work

Higher-order Charity suggests at least two possible topics of research:

1. Now that we may express classes and objects in Charity, we may also wish to incorporate other standard features of object-oriented programming. Most noteworthy is *inheritance* and the construction of a class hierarchy.

2. One would like to write Charity programs with *graphical user interfaces*. Interface entities such as windows and buttons are usually programmed as objects in modern programming languages. Now that Charity supports objects, one should be able to define window and button objects, assign them a graphical representation, and interact with them.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*, chapter 1–4. Springer, 1996.

[2] H. P. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. North Holland, revised edition, 1984.

[3] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.

[4] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1988.

[5] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[6] Robin Cockett and Dwight Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings. AMS, 1992.

[7] Robin Cockett and Dwight Spencer. Strong categorical datatypes II: A term logic for categorical programming. (to appear), May 1992.

[8] Roy L. Crole. *Categories for Types*. Cambridge University Press, 1993.

[9] Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

[10] Mike Hermann. A lazy graph reduction machine for Charity: CHarity Abstract Reduction Machine (CHARM). (unfinished), July 1992.

[11] Paul Hudak, Simon Peyton Jones, Philip Wadler, et al. Report on the programming language Haskell: A non-strict, purely functional language. *SIGPLAN Notices*, 27(5), May 1992.

[12] P. Katis, N. Sabadini, and R. F. C. Walters. The bicategory of circuits. In C. Barry Jay, editor, *Proceedings of Computing: the Australian Theory Seminar*, pages 89–108, December 1994.

[13] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA91: Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.

[14] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Functional Programming Languages and Computer Architecture*, 1995.

[15] Erik Meijer and Johan Jeuring. Merging monads and folds for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 228–266. First International Spring School on Advanced Functional Programming Techniques, Springer, December 1995.

[16] Robin Milner and Mads Tofte. *Commentary on Standard ML.* The MIT Press, 1991.

[17] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[18] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice Hall, 1986.

[19] D. E. Rydeheard and R. M. Burstall. *Computational Category Theory*. Prentice Hall International Series in Computer Science. Prentice Hall, 1988.

[20] Dwight L. Spencer. *Categorical Programming with Functorial Strength*. PhD thesis, The Oregon Graduate Institute of Science and Technology, January 1993.

[21] Charles Tuckey. The implementation of pattern matching in Charity, April 1994. Undergraduate honours thesis, Department of Computer Science, The University of Calgary.

[22] Charles Tuckey. The implementation of pattern matching in Charity. Master's thesis, The University of Calgary, July 1997.

[23] David Turner. An overview of Miranda. *SIGPLAN Notices*, December 1986.

[24] Peter Vesely. Typechecking the Charity term logic, April 1997. (documentation, `http://www.cpsc.ucalgary.ca/projects/charity/home.html`).

[25] Peter M. Vesely. Categorical combinators for Charity. Master's thesis, The University of Calgary, November 1996.

[26] R. F. C. Walters. *Categories and Computer Science*. Number 28 in Cambridge Computer Science Texts. Cambridge University Press, 1991.

[27] Dale Barry Yee. Implementing the Charity abstract machine. Master's thesis, The University of Calgary, September 1995.

Appendix A

Syntax

This appendix gives a formal definition of higher-order Charity syntax, type-theoretically. It is based on a first-order core term logic type theory by Peter Vesely [25], and was extended to a higher-order extended term logic type theory by Charles Tuckey [22]. The figures presented here are those of [22] minus completeness information.

The type theory is spread over three figures: figure A.1 deals with terms and their types, figure A.2 deals with patterns and their types, and figure A.3 combines the two, yielding functions and their types signatures.

variable	$$\frac{\Gamma \vdash \varphi \quad v \in \text{var}(T) \quad v \text{ not in } \Gamma}{\Gamma\,,\, v : T \vdash v : T}$$
unit	$$\frac{}{\vdash () : 1}$$
pair	$$\frac{\Gamma \vdash t_0 : T_0\,,\; t_1 : T_1}{\Gamma \vdash (t_0, t_1) : T_0 \times T_1}$$
record	$$\frac{\Gamma \vdash \begin{array}{ll} t_i : F_i(A, R(A))\,, & i = 1 \ldots r \\ g_j : E_j(A) \to F_j(A, R(A)) & j = r+1 \ldots n \end{array}}{\Gamma \vdash (d_i : t_i, d_j : g_j) : R(A)}$$
application	$$\frac{\Gamma \vdash t : S\,,\; f : S \to T}{\Gamma \vdash f\ t : T}$$

Figure A.1: Term Type Theory

variable	$$\frac{\Gamma \vdash \varphi \quad v \in \text{var}(T) \quad v \text{ not in } \Gamma}{\Gamma, v : T \vdash \varphi}$$
unit	$$\frac{\Gamma \vdash \varphi}{\Gamma\,,\,() : 1 \vdash \varphi}$$
constr	$$\frac{\Gamma\,,\,p_i : E_i(A, L(A)) \vdash \varphi \quad c_i \in \text{cstr}(L)}{\Gamma\,,\,c_i\,p_i : L(A) \vdash \varphi}$$
pair	$$\frac{\Gamma\,,\,p_1 : S\,,\,p_2 : T \vdash \varphi}{\Gamma\,,\,(p_1, p_2) : S \times T \vdash \varphi}$$
record	$$\frac{\Gamma\,,\,p_i : F_i(A, R(A)) \vdash \varphi \quad \begin{array}{l} v_j \in \text{var}(E_j(A) \to F_j(A, R(A))) \\ v_j \text{ not in } \Gamma \\ i = 1 \ldots r \\ j = r+1 \ldots n \end{array}}{\Gamma\,,\,(d_i : p_i, d_j : v_j) : R(A) \vdash \varphi}$$
int	$$\frac{\Gamma \vdash \varphi \quad i, j \in \mathbf{Z} \cup \{-\infty, \infty\} \quad i \leq j}{\Gamma\,,\,i..j : \textit{int} \vdash \varphi}$$
char	$$\frac{\Gamma \vdash \varphi \quad c_1, c_2 \in \text{ascii} \quad c_1 \leq c_2}{\Gamma\,,\,c_1..c_2 : \textit{char} \vdash \varphi}$$
pattern abstraction	$$\frac{\Gamma\,,\,p_i : S \vdash t_i : T \quad i = 1 \ldots m}{\Gamma \vdash p_i \mapsto t_i \mathrel{\vdots} S \to T}$$

Figure A.2: Pattern Type Theory

structor

$$\frac{\Gamma \quad s : S \to T \in \text{cstr} \cup \text{dstr}}{\Gamma \vdash s : S \to T}$$

defined function

$$\frac{\Gamma \vdash g_i \mathrel{|\!:} S_i \to T_i \quad f \in \text{comb}([S_i \to T_i]\,,\, S \to T) \quad i = 1 \ldots m}{\Gamma \vdash f\{g_i\} : S \to T}$$

case

$$\frac{\Gamma \vdash g \mathrel{|\!:} S \to T}{\Gamma \vdash \{g\} : S \to T}$$

fold

$$\frac{\Gamma \vdash g_i \mathrel{|\!:} E_i(A, T) \to T \quad c_i \in \text{cstr}(L) \quad i = 1 \ldots n}{\Gamma \vdash \{\!|\, c_i : g_i \,|\!\} : L(A) \to T}$$

map

$$\frac{\Gamma \vdash \begin{array}{ll} g_i^+ \mathrel{|\!:} A_i \to B_i, & i = 1 \ldots p \\ g_j^- \mathrel{|\!:} B_j \to A_j, & j = p+1 \ldots q \\ g_k^+ \mathrel{|\!:} A_k \to B_k, & k = q+1 \ldots r \\ g_k^- \mathrel{|\!:} B_k \to A_k & T \in \text{types} \end{array}}{\Gamma \vdash T \left\{ \begin{array}{c} g_i^+, \\ g_j^-, \\ g_k^+ \;\&\; g_k^- \end{array} \right\} : T(A) \to T(B)}$$

unfold

$$\frac{\Gamma\,,\, p_k : S \vdash \begin{array}{l} t_i^k : F_i(A, R(A))\,, \\ g_j^k \mathrel{|\!:} E_j(A) \to F_j(A, R(A)) \end{array} \quad \begin{array}{l} k = 1 \ldots m \\ i = 1 \ldots r \\ j = r+1 \ldots n \end{array}}{\Gamma \vdash \left(\!\left| p_k \mapsto \left| \begin{array}{l} d_i : t_i^k \\ d_j : g_j^k \end{array} \right. \right|\!\right) : S \to R(A)}$$

Figure A.3: Function Type Theory

Appendix B

A Catalogue of Charity Datatypes

This appendix lists the most common Charity datatypes, higher-order and otherwise. We begin with the *builtins* (section B.1), then user-definable *inductive* (B.2) and *coinductive* (B.3) datatypes, and briefly *type aliases* (B.4).

B.1 Builtin Datatypes

- The **nullary product** datatype: 1. Value:

$$()$$

- The **binary product** datatype: $\_ \times \_$. Value format:

$$(t_0, t_1)$$

- The **boolean** datatype:

$$\textbf{data}\ bool \longrightarrow C = \mathit{false} \mid \mathit{true} : 1 \longrightarrow C.$$

- The **integer** datatype: int (machine-level builtin). Values:

$$\ldots, -2, -1, 0, 1, 2, \ldots$$

- The **character** datatype: $char$ (machine-level builtin). Example values:

$$\backslash\text{cA ('A')}, \backslash\text{cB ('B')}, \backslash\text{cC ('C')}, \ldots$$

B.2 Inductive Datatypes

- The **coproduct** datatype:

$$\textbf{data}\ coprod(A,B) \longrightarrow C = \left| \begin{array}{lcccc} b_0 & : & A & \longrightarrow & C \\ b_1 & : & B & \longrightarrow & C. \end{array} \right.$$

- The **success-or-failure** datatype:

$$\textbf{data}\ SF\ A \longrightarrow C = \left| \begin{array}{lcccc} \mathit{ff} & : & 1 & \longrightarrow & C \\ ss & : & A & \longrightarrow & C. \end{array} \right.$$

- The **natural-number** datatype:

$$\textbf{data}\ nat \longrightarrow C = \left| \begin{array}{lcccc} zero & : & 1 & \longrightarrow & C \\ succ & : & 1 & \longrightarrow & C. \end{array} \right.$$

- The **list** datatype:

$$\textbf{data}\ list\ A \longrightarrow C = \left| \begin{array}{lcccc} nil & : & 1 & \longrightarrow & C \\ cons & : & A \times C & \longrightarrow & C. \end{array} \right.$$

- The **binary tree** datatype:

$$\textbf{data}\ bTree(A,B) \longrightarrow C = \left| \begin{array}{lcll} leaf & : & A & \longrightarrow & C \\ node & : & B \times (C \times C) & \longrightarrow & C. \end{array} \right.$$

B.3 Coinductive Datatypes

- The **colist** datatype:

$$\textbf{data}\ C \longrightarrow colist\ A = delist : C \longrightarrow SF(A \times C).$$

- The **infinite list** datatype:

$$\textbf{data}\ C \longrightarrow inflist\ A = \left| \begin{array}{lcccc} head & : & C & \longrightarrow & A \\ tail & : & C & \longrightarrow & C. \end{array} \right.$$

- The **exponential** datatype:

$$\textbf{data}\ C \longrightarrow exp(A,B) = fn : C \longrightarrow A \Rightarrow B.$$

- The **process** datatype:

$$\textbf{data}\ C \longrightarrow proc(A,B) = pr : C \longrightarrow A \Rightarrow C \times B.$$

B.4 Type Aliases

- The **string** datatype:

$$\textbf{data}\ string = list\ char.$$

Appendix C

The Implementation

Higher-order Charity is implemented. As of the time of writing the version is 1.9 (alpha) of June 1997. The interpreter consists of about 30000 lines of C code.

Charity is installed locally and may be invoked at a shell prompt by typing:

```
charity
```

It is available world-wide, together with literature, examples, etc., via the Charity homepage:

```
http://www.cpsc.ucalgary.ca/projects/charity/home.html
```

The Charity Development Group may be contacted at:

```
charity@cpsc.ucalgary.ca
```

Appendix D

A Simple Parser

This appendix gives a simple expression calculator whose scanning and parsing phases make heavy use of higher-order datatypes. This example is due to Schroeder and Cockett. The calculator takes a string involving binary addition and multiplication of numbers $0, 1, 2, \ldots$ (as well as optional white space), and returns "success-or-failure" of an integer result, respecting operator precedence. For example:

```
Charity>> calculate "2 + 10 * 4".
ss(42) : SF(int)
Charity>>
```

The Marc Schroeder's `calculator` code is:

```
rf "PRELUDE.ch".          % THE BASIC ENVIRONMENT
rf "syntax-trees.ch".     % ROBIN COCKETT'S PARSING UTILITIES


(*
 *  THE SCANNER
 *
 *)
```

```
data lex_states -> C = s0 | s1: 1 -> C.     % SCANNER STATES s0, s1


def char2int: char -> int     % CONVERT '0' TO 0, ETC...

    = c => { \c0..\c9 => sub_int (code c, code \c0)
           | _        => 0
           }
             c.


data tokens A -> C = PLUS : 1 -> C      % TOKENS FOR NATURAL NUMBERS, +, AND *
                   | TIMES: 1 -> C
                   | NUM  : A -> C.


def lex: 1 -> rS (char, list tokens int)     % A SCANNING AUTOMATON

    = () => (| (s0, (f, _)) => tok: c =>

             { \d32 => MORE (s0, (f, 0))
             | \c+  => MORE (s0, ((fn: l => fn (cons (PLUS, l),  f)), 0))
             | \c*  => MORE (s0, ((fn: l => fn (cons (TIMES, l), f)), 0))
             | \c0..\c9
                    => MORE (s1, (f, char2int c))
             | _    => FAIL
             }
               c

             |                end: ss fn ([], f)

             | (s1, (f, s)) => tok: c =>

             { \d32 => MORE (s0, ((fn: l => fn (cons (NUM s, l), f)), 0))
             | \c+  => MORE (s0, ((fn: l => fn (cons (NUM s, cons (PLUS, l)), f)), 0))
             | \c*  => MORE (s0, ((fn: l => fn (cons (NUM s, cons (TIMES, l)), f)), 0))
             | \c0..\c9
                    => MORE (s1, (f, add_int (mul_int (s, 10), char2int c)))
             | _    => FAIL
             }
               c
```

```
             |                 end: ss fn ([NUM s], f)
             |)
                (s0, ((fn: l => l), 0)).


def scan: string -> SF list tokens int     % THE SCANNER PROPER

    = s => p0 PARSE (lex, s).


(*
 *  THE INTERNAL REPRESENTATION
 *)


data expr -> C = Add: C * C -> C
               | Mul: C * C -> C
               | Val: int   -> C.


(*
 *  THE PARSER
 *)


data syn_states -> C = ps0 | ps1: 1 -> C.     % PARSER STATES ps0, ps1


% UPDATE AN INCOMPLETE EXPRESSION TREE WITH A NEW SUBTREE:

def update_expr: tokens int * expr * exp (expr, expr) * tokens int ->
                 tokens int * expr * exp (expr, expr)

    = (((op, val), f), op') =>

       { (PLUS,  PLUS)  => ((PLUS,  val), (fn: e => Add (fn (val, f), e)))
       | (PLUS,  TIMES) => ((TIMES, val), (fn: e => fn (Mul (val, e), f)))
       | (TIMES, PLUS)  => ((PLUS,  val), (fn: e => Add (fn (val, f), e)))
       | (TIMES, TIMES) => ((TIMES, val), (fn: e => fn (Mul (val, e), f)))
       | _              => ((op, val), f)
```

```
       }
        (op, op').

def syn: 1 -> rS (tokens int, expr)     % A PARSING AUTOMATON

    = () => (| (ps0, ((op, val), f)) => tok:

                  NUM i => MORE (ps1, ((op, Val i), f))
                | _     => FAIL

             |                          end: ff

             | (ps1, ((op, val), f)) => tok:

                  PLUS  => MORE (ps0, update_expr (((op, val), f), PLUS))
                | TIMES => MORE (ps0, update_expr (((op, val), f), TIMES))
                | _     => FAIL

             |                          end: ss fn (val, f)

             |) (ps0, ((TIMES, Val 0), (fn: e => e))).

def parse: list tokens int -> SF expr     % THE PARSER PROPER

    = l => p0 PARSE (syn, l).

(*
 *  THE EVALUATOR
 *)

def eval: expr -> int

    = e => {| Add: p => add_int p
            | Mul: p => mul_int p
            | Val: i => i
            |}
              e.
```

```
(*
 *  THE ENTIRE SYSTEM
 *)


def calculate: string -> SF int

    = s => SF{eval} compose_SF{scan, parse} s.
```

Robin Cockett's `syntax-trees` utility code is:

```
(*
 *    Parsing using "recursive syntax diagrams"
 *    Author:  Robin Cockett
 *    Date: 25 Sept '96
 *
 *    Recursive syntax diagrams were invented and used by Wirth to
 *    write the syntax of Pascal.  The way they are defined here
 *    guarantees an LL(1) (one token look ahead grammar)
 *    and it allows a very simple implementation.  Attributes
 *    can be added quite easily after the fact ....
 *)

%  Some utilities

def Id = x => x.

def foldleft{h: C * A -> C}: C * list(A) -> C
    = (c,L)  => fn(c, {|  nil:   ()  => (fn: x => x)
                       | cons: (a,f) => (fn: x => fn(h(x,a),f) )
                       |} L ).

def tail: list(A) -> list(A)
    = nil => nil
    | cons(_,L) => L.


%   Data structures for recursive syntax diagrams
%   After a token is taken in various things can happen
```

```
%     (1) The parser can FAIL
%     (2) The parser can decide the token is meant for the next
%         parsing step and it can PASS the token and any structures
%         it has built forward.
%     (3) The parser can eat a token and continue asking for MORE
%     (4) It can recursively call a substructure before continuing
%         with the parse.
%   At any stage in the parse one can ask supply a token (the tok
%   destructor) or see whether one can legally end (the end destructor
%   indicates the final states).

data FOLLOW(A,R,S) -> C =  PASS: A * R -> C
                        |  FAIL: 1 -> C
                        |  MORE: S -> C
                        | RMORE: S * exp(R,S) -> C.

%   Recursive syntax trees
data C -> rS(A,R) = tok: C -> A => FOLLOW(A,R,C)
                  | end: C -> SF(R).

%  An attribute recursive syntax diagrams allows an input attribute to
%  transform the diagram (inherited attributes and synthesized attributes
%  handled in this manner).

data C -> rSA(A,R) = rsa: C -> R => rS(A,R).

%  Two basic attribute recursive syntax diagrams
%  Failure (always fails)
def rSA_NULL: 1 -> rSA(A,R)
    = () => (rsa:_ => (tok: _ => FAIL,end: ff)).

%  Pass (always passes)
def rSA_PASS: 1 -> rSA(A,R)
    = () => (rsa:r => (tok: a => PASS(a,r), end: ss r)).


%  Sequencing recursive syntax tree generators:
%     When the syntax tree of one ends it passes the last
%     token and the result it is building to the next syntax tree
%     generator.
```

```
data SUM(A,B) -> C = b_0: A -> C
                   | b_1: B -> C.

def seq: rSA(T,R) * rSA(T,R) -> rSA(T,R)
  = (p,q) => (rsa: r =>
      (| b_0 t => tok: a =>
                { PASS(a',r') => FOLLOW{Id,Id&Id,b_1} tok(a',rsa(r',q))
                | MORE t' => MORE b_0 t'
                | RMORE(t1,c) => RMORE(b_1 t1,(fn: r => b_0 fn(r,c)))
                | FAIL => FAIL
                } tok(a,t)
       |          end: flatten_SF SF{ r => end rsa(r,q) } end t
       | b_1 t => tok: a => FOLLOW{Id,Id&Id,b_1} tok(a,t)
       |          end: end t
       |) b_0 rsa(r,p)
             ).

%  Kleene's star operator: repeating an attribute recursive syntax tree.
%  When the tree passes a symbol immediately it is done!

def star: rSA(T,R) -> rSA(T,R)
    = q => (rsa: r =>
        (| t  =>   tok: a => { PASS(a',r') => tok(a',rsa(r',q))
                             | z => z
                             } tok(a,t)
         |         end: end t
         |) rsa(r,q) ).

%  Alternating over recursive syntax trees:
%  If the first syntax tree fail immediately the second is used.

def alt: rSA(T,R) * rSA(T,R) -> rSA(T,R)
    = (p,q) => (rsa: r => (tok: a => {FAIL => tok(a,rsa(r,q))
                                     | z => z
                                     } tok(a,rsa(r,p))
                          ,end: {ff => end rsa(r,q)
                                | z => z
                                } end rsa(r,p)
                          )
              ).
```

```
%  Parsing using an attribute recursive syntax tree:
%      The parsing uses a stack of recursive syntax tree generators which
%      it develops every time it hits a recursive sub-syntax-diagram and
%      pops every time it finishes a parse pushed by a recursive
%      sub-syntax-diagram.

%    This routine presents the PASSed values to each member of the stack
%    until one of the generators does not simply pass it on ...

def use_stack
    = ((a,r),St) => { (MORE T',St) => ss(T',St)
                    | (RMORE(T,fT),St) => ss(T,cons(fT,St))
                    | _ => ff }
          foldleft{ ((PASS(a,r),St),c) => (tok(a,fn(r,c)),tail St)
                  | (z,_)               => z
                  } ((PASS(a,r),St),St).

%     This routine checks that a parse has successfully ended:
%     to do this we must check that what remains on the stack agrees
%     that the current state is a successful end point!

def end_stack
    = (T,St) =>
          foldleft{ (ss r,c) => end fn(r,c)
                  |    _     => ff
                  } (end T,St).

%    The parsing uses a stack (St) to hold the "continuation" syntax trees

def PARSE: rS(A,R) * list(A) -> SF(R) * list(A)
    = (T,L) => { (ss(T,St),L)  => (end_stack(T,St),L)
               | (_, L) => (ff,L) }
   foldleft{ ((ss(T,St),L),a) => { MORE T' => (ss(T',St),tail L)
                                 | RMORE(T,fT) => (ss(T,cons(fT,St)),tail L)
                                 | PASS(a,r) => { ss x => (ss x,tail L)
                                                | ff => (ff,L)
                                                } use_stack((a,r),St)
                                 | FAIL => (ff,L)
                                 } tok(a,T)
           | ((ff,L),a) => (ff,L)
```

```
} ((ss(T,[]),L),L).
```

The Charity Standard Prelude (`PRELUDE.ch`) is a basic environment of common datatypes and functions.

Index